Formal Verification for Feature-based Composition of Workflows

Stephan Adelsberger[1], Bashar Igried[2],
Markus Moser[1], Vadim Savenkov[1], and Anton Setzer[3]

[1] Dept. of Information Systems, Vienna University of Economics, Austria
[2] Faculty of Information Technology, The Hashemite University, Jordan
[3] Dept. of Computer Science, Swansea University, Swansea, UK

Abstract. We developed *FeatureAgda*, a framework to specify and prove properties of feature-based composition of workflows implemented in the Feature-Oriented Software Production Lines paradigm. The resulting workflows allow for adaptation at runtime by changing the set of enabled features. Our framework allows the modular definition of features and promotes the separation of concerns in the workflow definitions. In addition, we obtain a single artefact that represents the entire software product line through the use of the expressiveness of dependent types, allowing the application of family-level formal verification.
Our framework is based on Agda which is both a theorem prover and a programming language. We apply our framework to a case study from the healthcare domain which implements feature-based composition of workflows for medication prescriptions. Our setting allows the workflow to be changed according to patients' specific cases and doctors' needs while having trustworthiness through formal verification.

Keywords: feature-oriented software development·product-Based Workflows ·Agda·theorem proving·dependable software·family-level formal verification ·verification of workflows·formal verification

1 Introduction

One of the most significant challenges of software design processes is resilience to changes, which is especially true for dependable systems where verification infrastructure often adds considerable complexity to the software design process and where even small modifications might have far-reaching implications for the verification process. In this paper we address the current lack of flexibility of rigorously designed systems based on a *software product lines* (SPL) paradigm. Specifically, our contribution is a novel verification framework for *feature-oriented software product lines* [3], which we named *FeatureAgda*.

Software product lines allow the assembly of software-intensive systems based on a predefined set of features using reusable software components [3]. Areas of application include the automotive sector, operating systems/kernels and the healthcare domain [16].

Currently, there are several design choices for SPLs and their verification. The first design choice is the feature binding time: static versus dynamic/runtime. The second choice is the implementation: either compositionally by implementing features as modules that can be added or not to a product, or annotatively by embedding variation points corresponding to different features directly. Finally, there is product-based verification of specific feature configurations versus family-based formal verification of the whole SPL [27].

Essentially, compositional implementation puts *flexibility first* and postulates that the systems should be developed from modular components. However, regarding the formal verification and analyzability, annotation-based approaches are traditionally better suited for family-based verification and can be considered to prioritise *safety first*. We consider family-based verification a safer option for run-time feature binding time, since we don't want to call any theorem provers at run-time but use statically guarantees provided by family-based verification for any feature selection.

With *FeatureAgda*, our goal is to cater for both design styles. Both the configuration and the verification phases support dynamic composition of modules at run-time. From a pure software design perspective this approach might be seen as fairly trivial, since most software tools nowadays support some degree of extensionability (via plugins or scripting). From the verifiability perspective, however, our approach is novel, ensuring that specifications of modules (readily wired together or anticipated at runtime) fit together and support *reasoning about features*. For instance, one can formulate claims that hold true in all configurations based on features with some properties, without the necessity to fully specify those configurations upfront.

In this way, in *FeatureAgda* one can compose systems both statically and dynamically (even in the process of system execution) and still enjoy the benefits of formal verification provided by the formal framework built atop of a theorem prover with dependent types support (our prover of choice in this paper is Agda [2]).

We validate our approach using a case study from the healthcare domain, namely the workflow of medication prescription in a complex and high-risk setting of prescribing anticoagulants, commonly also called blood thinners. The recent introduction of a highly efficient class of so-called *novel anticoagulants* (NOACs) has made the prescription process exceedingly complex, since many parameters must be considered for the correct drug and dosage selection. Despite detailed guidelines and special measures taken in most hospitals to reduce the probability of errors (which can often be life-threatening), an estimated 16% of prescription errors [29] and up to 60.8% of dosage errors still currently occur in the everyday practice [9].

The constraints of the NOAC prescription process are not limited to the correct interpretation of multiple medication leaflets provided by the manufacturers under the control of the European Medical Association (EMA). As new medications are introduced to the market and the new research results become available, the EMA prescription regulations and medication leaflets are adapted, typically on a yearly basis. An additional complexity comes from the requirement to comply with the policies of insurance providers that cover the cost of the medication. According to many healthcare providers' policies (for instance the Austrian social insurance regulations), the cheapest medication among equally effective ones should be prescribed. As

the market prices fluctuate, this requirement effectively means that the prescription procedure should be adapted continuously.

Last but not least, the same body of official recommendations, guidelines and policies are normally implemented differently depending on the hospital, and even show variations between different departments within the same institution. In fact, actual workflows of different departments consist of different sequences of steps. Hence, a competitive healthcare information system should cater for both the adaptability of high-level specifications to ensure compliance with the actual normative documents and the flexibility of implementation of actual workflows.

Our ongoing case study in the Vienna General Medical Hospital (AKH) tests a formally verified computer prescription assistant, guiding the doctors through the prescription workflow, including soliciting the patients' information through necessary medical examinations. The design of the underlying system was quite labor consuming, even in the basic case of implementing and verifying a single version of prescription guidelines. While the intermediate results of our study seem promising, adaptability has arisen as the next pressing issue: how can one cater for future policy changes and still retain the guarantees provided by the formal system design?

The present paper is an attempt to answer this question. The contributions that we made are as follows:

- *FeatureAgda as a framework implementing flexible SPLs using dependent types*, supporting modular description and implementation of features. An executable workflow is generated from a feature selection.
- *Support for static and dynamic (runtime) feature binding.* For example, in the NOAC prescription use case, both the support for verification and adaptability are intrinsic requirements, as discussed above.
- *Both specifications and proofs are variability-aware* in *FeatureAgda*. We support reasoning over features, such as a definition of feature properties and proving claims about the system based on those properties, without having the full specification of all the modules. In this way, we can support proofs which are parameterised by arbitrary feature configurations, known as *family-based deductive formal verification* [27] of SPLs.
- *An unbounded number of states and arbitrary IO.* Our specifications support an unbounded number of states and also support proofs over programs that include IO actions such as database queries.
- *Evaluation based on a relevant case from the healthcare domain.* NOAC selection proved to be an excellent showcase for the dependable software development methods, as an area where, on the one hand, formalization brings immediate benefits and, on the other hand, yielding advanced functionality such as adaptability. Our formalization of the NOAC prescription use case is publicly available and can be used for testing and benchmarking similar tools and methodologies in the future.

Source Code. All displayed Agda code has been extracted automatically from type-checked Agda code [1]. For readability, we have hidden some details and show only the crucial parts of the code. The full source code is available online.[4]

[4] `https://gitlab.com/Stefanad/FeatureAgda`

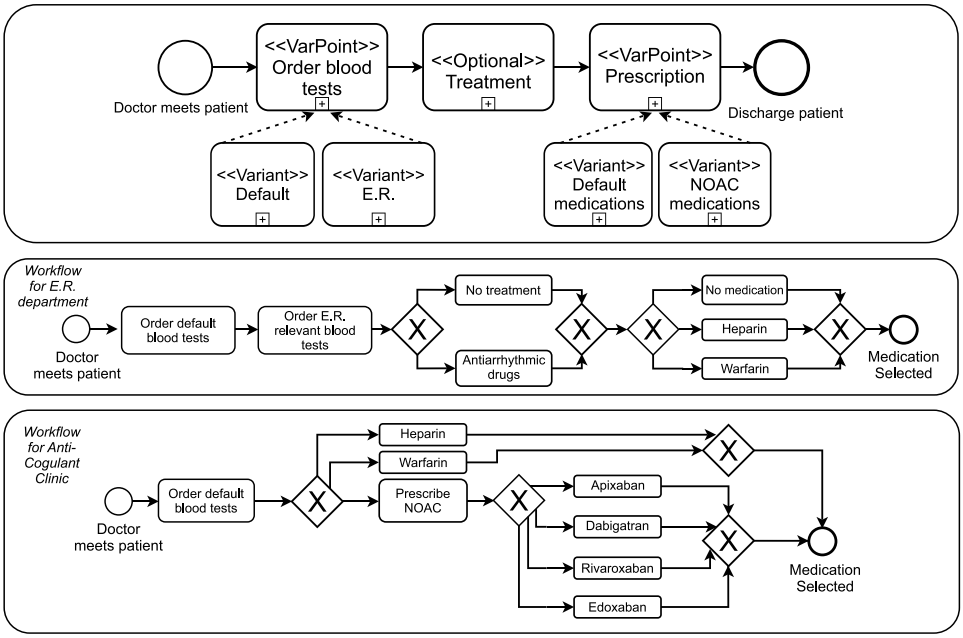

Fig. 1: A part of the prescription workflow

2 Background and Context

2.1 Software Product Lines

We use the term Software Product Lines (SPLs) [3] to denote the software design principle in which the software products are developed from a common set of core assets or artefacts with variability and reuse among the main principles. Feature-oriented SPLs place special emphasis on enabling modularity by formalizing the general notion of feature as an end-user visible characteristic of a software product, which vendors can compose to cater for requirements of a particular application domain.

In our running example, we apply the feature-oriented SPL approach to extend a *workflow* (or a business process whose actions are executed by humans) with *variability points*, as illustrated in Fig. 1.

The workflow in Fig. 1 is based on an ongoing case study evaluating the uses of dependable software in healthcare, which we are currently conducting in the AKH Hospital of Vienna. The workflow formalises the medical process of prescribing anticoagulant medications. For the sake of simplicity, we only present several essential steps of the actual workflow implemented in the hospital, including blood tests, treatment, and writing a prescription for the required medications.

The workflow in Fig. 1 is displayed in PESOA notation [25], which is an extension of the business process notation BPMN with additions for variability. For instance, the prescription activity is performed by doctors with different specializations. In an ER

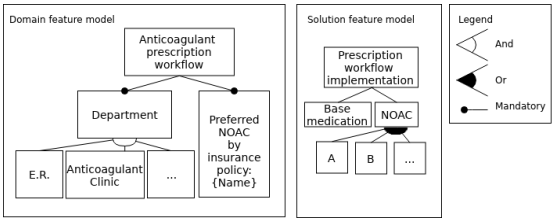

Fig. 2: A part of the feature model for the NOAC prescription workflow

department, only two "classical" anticoagulants are used, whereas in the anticoagulant clinic, novel anticoagulants (NOACs) are also applied.

The feature model for the prescription workflow is shown in Fig. 2 in the form of a tree where nodes represent features and edges depict the relationship between them. We choose a staged feature model [13], where a selection of user-relevant features on the left is mapped to a feature selection on the right that is more solution relevant. For example, on the right-hand side, we consider individual NOAC medications as features.

Domain features such as Preferred NOAC by insurance policy, Department are represented, among others, by the variants ER and Anticoagulant clinic. We model Preferred NOAC by insurance policy as an attributed feature to accommodate changes and updates to the prescription regulations (e.g. in combination with runtime feature selection). Preferred NOAC by insurance policy has preferred NOAC medication as an attribute, normally referring to the cheapest NOAC on the market, which constitutes a so-called attributed feature model [6].

The actions and features outlined in Fig. 1 and Fig. 2 are implemented in a software system developed in the dependently typed functional programming language Agda.

2.2 Agda

Agda [2] is a theorem prover and also as a dependently typed programming language. Types can depend on arbitrary values. This is in contrast to functional programming languages such as Haskell and ML that separate types and values. Dependent types are very handy for implementing and proving properties of SPLs, since any type can depend on the value of a particular feature configuration.

There are several levels of types in Agda, the lowest is for historic reasons not called "Type" but referred to as Set. The next level type is called Set_1, which has the same closure properties as Set but also contains Set as an element. The reason for the levels is to avoid Girard's paradox.

Agda has a termination and coverage checker. The coverage checker guarantees that the definition of a function covers all possible cases, and the termination checker verifies that definitions terminate. Without them, Agda would be inconsistent.

Types in Agda are given as dependent function types, inductive types and record types. A dependent function type is written as $(x : A) \to B$, which maps an element x of type A to an element of type B, where the type B may depend on x.

Inductive data types are dependent versions of algebraic data types as they occur in functional programming. Inductive data types are given as sets A together with

constructors. For instance, the collection of finite numbers (i.e., numbers smaller than a given limit) is given as a map from $\mathbb{N}$ to Set:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (succ n)
  suc :  {n : ℕ} → Fin n → Fin (succ n)
```

Here $\{n{:}\mathbb{N}\}$ is an implicit argument. Implicit arguments are omitted, provided they can be uniquely determined by the type checker. The elements of (Fin n) are those constructed from applying these constructors. Therefore, we can define functions that operate on (Fin n) by case distinction on these constructors using pattern matching (similar to pattern matching in Haskell).

Record types are used to describe the grouping of several categories into one type, for example:

```
record AB : Set where
  a : ℕ
  b : Fin a
```

The above defines a new record type AB with two fields. The first field is a, which has type $\mathbb{N}$ and the second field is b, with type Fin a. Also, Agda allows dependent record type where the type of one field depends on other fields. In the above example, the type of b depends on the value of a.

Agda has a mechanism for defining infix operators, where the arguments of infix operators are denoted by the underscore (_). For example, disjunction which is infix on truth value can be defined as follows:

```
_or_ : Bool → Bool → Bool
false or m = m
true  or m = true
```

Note that in our code examples we sometimes show only the type of a function and omit for brevity the full implementation.

3 Workflow Specification in *FeatureAgda*

Our main concern is the solution space and the aspect of formal verification of SPLs. We don't consider additional aspects of SPLs in this paper, such as requirement analysis or documentation of SPLs. In the following, we give an overview of our framework from the perspective of a user of the system.

We implement features in our compositional SPL approach using functions as our compositional units. Thus, a feature implementation is a function that maps a

product to an extended/adapted product. In our workflow case, this is as a map from workflows to extended/adapted workflows:

```
featureImplementation : Workflow → Workflow
```

We will later show the dependently typed version of this function in full detail. A specific workflow product can be expressed via function composition. Assuming `trivialFeatureImpl : Workflow`, we could express the composition of the blood-test (sub) workflow (cf. Fig. 1) for the E.R. department as follows:

```
trivialWorkflow          : Workflow
defaultOrderBloodTests : Workflow → Workflow
orderBloodTestsER        : Workflow → Workflow

workflowBloodTestER : Workflow
workflowBloodTestER = orderBloodTestsER (defaultOrderBloodTests trivialWorkflow)
```

A further step is to generate an executable workflow version according to such defined workflows. The result is an executable program that implements the different graphical user interface (GUI) forms necessary to guide a doctor during the prescription workflow. We have implemented a generic function that converts workflows to executable programs:

```
compileWorkflowToProgram : Workflow → ExecutableIOProgram

main : ExecutableIOProgram
main = compileWorkflowToProgram workflowBloodTestER
```

The final step is to reason about and formally verify executable workflows. For such proofs, we define finite simulations of workflows. As mentioned, a major goal is the verification of family-based proofs. This allows us to quantify over `featureImplementation` and feature selections and prove that after composing a feature implementation with another workflow, all relevant safety specifications still hold. In the context of prescription workflows, safety usually means the patient is prescribed a save medication together with a correct dose.

Experience has shown that functions as units of composition are flexible and modular. However, it is challenging to allow for a high degree of expressiveness of the feature adaptions, especially, while maintaining (family-based) analyzability based on formalised proofs. Our solution to this challenge is the topic of the next chapter.

4 Workflow Verification and Implementation

We represent workflows as state machines embedded in dependent type theory within Agda. Typical in healthcare, workflows depend on a lot of data (patient data, blood test results, etc.) and also need to interface with databases and diagnosis machines (e.g., Electrocardiography machines).

Here, state machines may have an unbounded number of states and allow for arbitrary IO interactions, so they are more expressive than finite state machines.

4.1 State machines

‘

We define our workflow state machines generically. We say a workflow state has a view that represents the workflows state to a user (in our case doctors). We assume two abstract types View and UserInput. The latter is a map from a view to a type representing the user input for that view.

To define state machines for workflows we first define a handler for the user input:

```
machineInputHandler : (S : Set)(v : View) → Set
machineInputHandler State v = (input : UserInput v) → IO State
```

A handler is a function that maps user input to IO programs that calculate a successor state for the state machine. A handler takes a type of states S, a view of type View and finally maps the user input (dependent on the view) to IO programs of type S. The latter means the IO programs return a value of type S, where the return value represents the new state.

A MachineState is a record definition associating a view with a handler (called handle) for user input for that view. The definition is as follows:

```
record MachineState (State : Set) : Set where
  view : View
  handleUserInput : machineInputHandler State view
```

Note that this is a dependent record, as handle depends on view. Here "type safety" is already a much stronger consistency property than most existing approaches, as the type system statically ensures that each user input is properly handled in the state machine.

The record is generic, as it is parametrised over the state of simple states S.

Finally, we can define a StateMachine as a mapping from simple states to MachineState (with associated views and handlers):

```
StateMachine : Set → Set
StateMachine State = (s : State) → MachineState State
```

4.2 Extending State Machines with Features

Features can be used to modify the structure and content of a state machine. For example, a feature might add a new transition between states s and s' triggered by user input that is added to the view associated with state s. Each feature may be included or not in the final workflow application yielding a family or product line of possible workflow applications depending on which features are included.

Generic feature-oriented state machines are functions which for each feature yield a state machine for the set of states.

```
FeatureMachineNaive : (F S : Set) → Set
FeatureMachineNaive F S = (f : F) → StateMachine S
```

Here $(F\ S : \mathsf{Set}) \to \mathsf{Set}$ stands for $(F : \mathsf{Set})\,(S : \mathsf{Set}) \to \mathsf{Set}$ which in turn stands for $(F : \mathsf{Set}) \to (S : \mathsf{Set}) \to \mathsf{Set}$.

When expanding features, we will add extra states to the machine. We want to modify the original states independently of the new states added. Therefore, we separate the set of states into two sets, namely the set B of base states, common to all machines independent of features added, and the set S of extra sets.

$$\begin{aligned}&\mathsf{FeatureMachine} : (F\ B\ S : \mathsf{Set}) \to \mathsf{Set}\\&\mathsf{FeatureMachine}\ F\ B\ S = (f : F) \to \mathsf{StateMachine}\ (B \uplus S)\end{aligned}$$

Depending on a feature, the resulting state machine consists of the disjoint union ($\uplus$) of the base states and the new states. This enables features to dynamically add new states in a monadic way. It also enables features to be applied to a machine multiple times. For example, given a feature that adds a button to the GUI associated with a state, applying that feature twice would add two buttons to that GUI.

We can extend a feature machine by a new state, provided we give the information how to handle the new state (which will yet be unreachable from the previous states):

$$\begin{aligned}&\mathsf{addStateToFeatureMachine} : \{F\ B\ S' : \mathsf{Set}\}\ (fm : \mathsf{FeatureMachine}\ F\ B\ S')\\&\qquad(new : \mathsf{MachineState}\ (B \uplus (S' \uplus \mathsf{Void})))\\&\qquad\to \mathsf{FeatureMachine}\ F\ B\ (S' \uplus \mathsf{Void})\end{aligned}$$

In the above definition $\{F\ B\ F' : \mathsf{Set}\}$ stand for three hidden arguments: they are like arguments $(F\ B\ F' : \mathsf{Set})$, but when using them they can be omitted, if Agda can automatically infer them. This helps to shorten the code. Void is the type without specific information, having only one trivial element triv.

We can add a new dummy feature:

$$\begin{aligned}&\mathsf{addFeatureToFeatureMachine} : \{F\ B\ F'\ S : \mathsf{Set}\}\\&\qquad(fm : \mathsf{FeatureMachine}\ F\ B\ S)\\&\qquad\to \mathsf{FeatureMachine}\ (F \times F')\ B\ S\end{aligned}$$

In order to give meaning to a new state and feature added, we need to adapt other states so that they are modified depending on new features added. We illustrate this by giving an example constructing a simple medical example:

We first construct a basic machine for the purpose of prescribing medication, with only the trivial feature Void and no extra states (given by state set $\emptyset$). The relevant blood test for the prescription is the estimation of the renal function, which is measured as the value of creatinin clearance (CrCl). A value below 15 means insufficient kidney function, for which Warfarin is the medication of choice. We give only the definition for the initial state enterCrCl, which, depending on whether the CrCl value is <15 or ≥ 15, goes to different states for prescribing medications:

$$\begin{aligned}&\mathsf{basicMachine} : \mathsf{FeatureMachine}\ \mathsf{Void}\ \mathsf{StatesBasic}\ \emptyset\\&\mathsf{basicMachine}\ f\ (\mathsf{position}\ \mathsf{enterCrCl}) =\\&\quad\mathsf{disjointChoiceState}\ \texttt{"CrCl < 15"}\ (\mathsf{position}\ (\mathsf{prescribeMedication}\ {<}15))\\&\qquad\qquad\texttt{"CrCl} \geq \texttt{15"}\ (\mathsf{position}\ (\mathsf{prescribeMedication}\ \geq 15))\end{aligned}$$

Now we add to this machine a new state for handling NOAC medications:

```
NoacMAddNewState : {F S : Set}(noa : NOAC)
                   → (fm : FeatureMachine F StatesBasic S)
                   → FeatureMachine F StatesBasic(S ⊎ Void)
```

and a new dummy feature allowing NOACS to be processed:

```
NoacMAddFeature : {F B S : Set}(noa : NOAC)(fm : FeatureMachine F B S)
                  → FeatureMachine (F × (FeatureNOAC noa)) B S
```

Now we adapt the state for prescribing the medication for renal value ≥ 15 by allowing the NOAC in question to be a choice for prescription, and keeping all other states as they were before:

```
NoacMAdaptFeature :
 {F S : Set}(noa : NOAC)
 (fm : FeatureMachine (F × (FeatureNOAC noa)) StatesBasic (S ⊎ Void))
 → FeatureMachine (F × (FeatureNOAC noa)) StatesBasic (S ⊎ Void)
NoacMAdaptFeature noa fm (f , yesNOAC .noa)
                         (position (prescribeMedication ≥15))
 = addChoice2State (fm (f , yesNOAC noa) (position (prescribeMedication ≥15)))
     (noac2Name noa) newState
NoacMAdaptFeature noa fm (f , selection) s = fm (f , selection) s
```

We can now add to a feature machine this new feature, by adding the previous operations in sequence:

```
NoacFeatureMachine noa fm = NoacMAdaptFeature noa
                            (NoacMAddNewState noa
                            (NoacMAddFeature noa fm))
```

We an as well add two NOACs by adding the above operation twice for the two NOACs in question:

```
NoacFeatureMachine2 noa1 noa2 =
 NoacFeatureMachine noa1 (NoacFeatureMachine noa2 basicMachine)
```

The resulting code can now be compiled into an executable GUI:

```
NoacFeatureMachine2GUI {F}{S} f noa sta fm =
 compile2GUI ((NoacFeatureMachine noa fm) (f , yesNOAC noa)) sta
```

We can now prove correctness theorems. For instance we show that if we add two NOACs to the basic machine, we will reach from the prescription state the state where Warfarin is prescribed in case the renal value is <15:

```
theoremWarfarin : ∀ (noa1 noa2 : NOAC) →
 prescribeMedicationState noa1 noa2 <15 -eventually-> warfarinState noa1 noa2
```

4.3 Calling Machines with Different Features in a Monadic Way

We will now show how one state-machine can call another state machine using different feature selections. We use this approach for dynamic (runtime) feature binding. In order to do this we first define a monadic extension of state machines and feature machines.

The theoretical basis for the use of monads in functional programming was laid by Moggi [23]. It was pioneered by Peyton-Jones and Wadler [28,24] as a paradigm for representing IO in functional programming, especially Haskell. An element of the IO monad ($\mathsf{IO}\,A$) is an interactive program which continuously interacts with the real world, and possibly terminates. If it terminates it gives a return value, an element of type A. The monadic approach allows for monadic composition: If we have one program $p : \mathsf{IO}\,A$ and a function $q : A \to \mathsf{IO}\,B$ we can form a program $r := p \gg q$ of type $\mathsf{IO}\,B$: Program r first executes program p. If p terminates, returning value $a : A$, then r continues executing $(q\,a)$. If that program terminates with return value $b : B$ then r terminates as well with the same return value. Monads allow therefore to compose programs in a modular way.

We apply the monadic approach to state machines as follows: The monadic version of a state machine has an extra argument A of return values returned by the state machine. Events are handled by an element of $(\mathsf{MachineState}\,(S \uplus A))$. Here $S \uplus A$ is the disjoint union of S and A, having elements $(\mathsf{inj}_1\ s)$ for $s : S$ and $(\mathsf{inj}_2\ a)$ for $a : A$. If the user triggers an event, an element of $(\mathsf{IO}\,(S \uplus A))$ is executed. If that IO program terminates with $(\mathsf{inj}_1\ s)$, the state machine continues in state s. If the IO program terminates with $(\mathsf{inj}_2\ a)$ then the state machine terminates with return value a. Monadic state machine and the monadic feature machine are defined as follows:

$$\begin{aligned}
&\mathsf{StateMachineMonadic} : (A\ S : \mathsf{Set}) \to \mathsf{Set}\\
&\mathsf{StateMachineMonadic}\ A\ S = S \to \mathsf{MachineState}\ (S \uplus A)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{FeatureMachineMonadic} : (A\ F\ S : \mathsf{Set}) \to \mathsf{Set}\\
&\mathsf{FeatureMachineMonadic}\ A\ F\ S = (f : F) \to \mathsf{StateMachineMonadic}\ A\ S
\end{aligned}$$

We can now take one state-machine $machine_1$ which has as return value a feature f and a state of a monadic feature machine $(machine_2\,f)$ for that feature. Assume the return values of $(machine_2\,f)$ are states of $machine_1$. Then we can combine both machines into one state-machine *combimachine*. It has as states the states from $machine_1$ and pairs consisting of a feature f for $machine_2$ and a state s of $machine_2$. The state machine *combimachine* operates as follows: In a state of $machine_1$, it executes as $machine_1$ until it terminates. Once $machine_1$ has terminated with return value (f, s), $machine_2$ is started with feature f and state s. That machine is executed, until it terminates, giving as return value a state s' of $machine_1$. The control flow continues again with $machine_1$ starting in state s'. The type of this operation is as follows:

$$\begin{aligned}
&\mathsf{combineStateFeatureMachine} : \{F\ S\ S' : \mathsf{Set}\}\\
&\quad (machine_1 : \mathsf{StateMachineMonadic}\ (F \times S')\ S)\\
&\quad (machine_2 : \mathsf{FeatureMachineMonadic}\ S\ F\ S')\\
&\quad \to \mathsf{StateMachine}\ (S \uplus (F \times S'))
\end{aligned}$$

As an example we take as first machine a machine which has only one state, in which it gives the user the choice between having the creatinin clearance (CrCl) value <15 or ≥ 15. In the first case it calls the feature machine for prescribing the medication having the NOAC feature deactivated, which means that there is no option of prescribing a NOAC. In the second case this feature is activated, allowing a NOAC prescription:

```
callMachine noa enterCrCl =
  disjointChoiceState "CrCl < 15"  (terminate (noNOAC noa , prescribeMed))
                      "CrCl ≥ 15" (terminate (yesNOAC noa , prescribeMed))
```

This machine will be extended to a function callMachineFeatured to allow for extra features and states. As feature machine we take the same feature machine as before, but in case of discharging the patient giving a return value which effectively calls the first state machine again. We omit as well the first state which is now handled by callMachineFeatured. We combine these two machines and obtain one machine: In that combined machine the first machine (which has no features) calls the second feature machine, and selects the feature used dynamically depending on its inputs. When the second feature machine has terminated it makes a call back to the first machine.

```
NoacMachine noa =
  combineStateFeatureMachine
  (callMachineFeatured triv noa)
  (NoacFeatureMachine noa (mapFeatureMach (λ _ → enterCrCl) basicMachine))
```

4.4 Case Studies Carried Out

We have applied the above approach to several paradigmatic examples. One example is a vending machine [11]. Possible features are adding the option to obtain specific beverages such as tea, coffee, or soda, the option of having a cancel body, and of giving change. We defined this in a modular way so that operators can be applied one by one to the machine. We defined as well an operator for adding a generic button for any beverage, which is given by a string. That operator can be applied arbitrarily many times to the vending machine, giving an unbounded number of buttons.

The second example was a simple ATM and similar to the Bank Account SPL [26]. The user can withdraw from the ATM as long as there is enough money left.

A third example allows GUI which have an unbounded number of buttons. Each button results in an extension of the GUI by adding a certain amount of buttons to the GUI. This example demonstrates that this framework allows to define GUIs which have infinitely many states, where the states correspond to GUIs which differ in the number of buttons.

Finally we developed a more advanced version of the NOAC medication prescription example. The caller machine collects data about the patient, and then calls a feature machine. The features of the feature machine form a subset of the NOAC medications available. The feature machine will then handle prescription of the medicines and then switch back to the state machine handling user input.

5 Related Work

Verification of Software Product Lines Our examples in Section 4.2 show how our library can be used to realise a feature-oriented software product line (SPL) [3] of workflow applications; that is, a workflow application that varies depending on which of several features are enabled. In general, SPLs can be developed either compositionally [5] by implementing features as modules that can be added or not to a product, or annotatively [19,15] by embedding variation points corresponding to different features directly in the source code. There are tradeoffs between the two styles [19]. Our realization of SPLs combines benefits of both approaches. Like compositional approaches, our approach supports the modular definition of features and promotes a separation of concerns. However, like annotative approaches we ultimately obtain a single artefact that represents the entire SPL, promoting family-level analyses [26].

There is a huge body of work on verifying properties of software product lines [14,10,11,20,22,27,26]. A strength of our implementation is that we can use Agda's theorem proving capabilities to prove arbitrary properties about SPLs. In particular, we can prove that software products and the product line itself are correct w.r.t a formal specification. This is in contrast to other approaches that verify only aspects such as the respective software contracts and (class) invariants. The work by Classen et al. [11], which introduces the vending machine example, uses model checking to verify several safety properties of the system, for example, that for any selection of features, all included states are reachable. Such properties can be formulated as types in Agda and proved by providing a value of that type.

Variability in Healthcare Process Models Asadi et al. [4] investigate how variability can be represented as customizations of reference process models. They present a case study modelling an information systems for the support of optometrists in their daily activities. Additionally, they provide a framework which is able to discover inconsistencies with regard to the reference process model automatically. However, they don't include any formalised proves such as that the process model variants are correct with regards to a formal specification.

Dynamic feature binding and GUI applications Kramer et al. [21] have contributed an approach that supports both static and runtime feature binding of GUIs based on Dynamic SPLs (DSPLs [17]). We also support runtime feature binding of executable workflows which include GUI forms. Kramer et al. [21] discuss certain challenges including keeping the components of Model-View-Controller (MVC) architectures consistent in the presence of variability. In this regard, note that in Section 4 we demonstrate that we can automatically solve this consistency problem with a dependent record type. For example, consider the definition of MachineState. This aspect highlights that dependent types can express properties generically, which is beneficial in realizing feature-oriented SPLs.

6 Conclusion

We have developed *FeatureAgda*, a dependable design of Feature-Oriented SPLs. Dependent types proved to be a powerful tool. They were essential, since the type of machineInputHandler depends on the View. Dependent types allowed to call different features of machines dynamically, allowing for run-time feature binding. This was facilitated by the use of monadic state machines and feature machines.

Our approach was implemented as a library in the Agda programming language. It should be noted however, that also other languages with dependent types support (e.g., Idris [8,18] or Coq [7,12]) could be used in a very similar fashion for the implementation.

Future work. Although proving claims about our experimental system implementation was not overly complicated, the verification step in Agda is still not a fully automated process. Increasing the degree of automation in this respect is thus a relevant future work task. Some applications such as our running example of NOAC prescriptions can be easily shown to be decidable and amenable to automatic verification. Our prototype implementation in the NOAC prescription context shows how properties should be formulated in Agda to allow for automatic type checking. We plan to further generalise and extend this to support verification of decidable processes in other domains as well.

Our ongoing work also includes the support for a larger subset of BPMN specification as a way of importing existing workflows into Agda and proving properties about them, thus extending our initial set of evaluation examples. This would in turn require support for richer functionality such as feature interactions, concurrency, roles and access right management.

References

1. Adelsberger, S., Igrid, B., Moser, M., Savenkov, V., Setzer, A.: Formal verification for feature-based composition of workflows (2018), `https://gitlab.com/Stefanad/FeatureAgda`, git respository
2. Agda Community: Agda documentation (2018), `http://agda.readthedocs.io/`
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer-Verlag, Berlin/Heidelberg (2013)
4. Asadi, M., Mohabbati, B., Gröner, G., Gasevic, D.: Development and validation of customized process models. Journal of Systems and Software 96, 73–92 (2014)
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. on Software Engineering (TSE) 30(6), 355–371 (2004)
6. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: International Conference on Advanced Information Systems Engineering. pp. 491–503. Springer (2005)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
8. Brady, E.: Type-driven Development with Idris. Manning Publications, Greenwich, Connecticut, 1 edn. (2017)
9. Buchholz, A., Ueberham, L., Gorczynska, K., Dinov, B., Hilbert, S., Dagres, N., Husser, D., Hindricks, G., Bollmann, A.: Initial apixaban dosing in patients with atrial fibrillation. Clinical cardiology 41(5), 671–676 (2018)
10. Chen, S., Erwig, M.: Type-based parametric analysis of program families. In: ACM SIGPLAN Int. Conference on Functional Programming (ICFP). pp. 39–51. ACM (2014)

11. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: Proc.of the 32nd Conf. on Soft. Eng. pp. 335–344. ICSE '10, ACM, New York, NY, USA (2010)
12. Coq Community: The Coq Proof Assistant (2018), https://coq.inria.fr/
13. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
14. d'Amorim, M., Lauterburg, S., Marinov, D.: Delta execution for efficient state-space exploration of object-oriented programs. IEEE Trans. on Software Engineering (TSE) 34(5), 597–613 (2008)
15. Erwig, M., Walkingshaw, E.: The Choice Calculus: A Representation for Software Variation. ACM Transactions on Software Engineering and Methodology (TOSEM) 21(1), 6:1–6:27 (2011)
16. Gomes, A.T.A., Ziviani, A., Correa, B.S.P.M., Teixeira, I.M., Moreira, V.M.: Splice: a software product line for healthcare. In: Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium. pp. 721–726. ACM (2012)
17. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41(4) (2008)
18. Idris Development Team: Idris. a language with dependent types (2018), https://www.idris-lang.org/
19. Kästner, C., Apel, S.: Virtual separation of concerns – a second chance for preprocessors. Journal of Object Technology 8(6), 59–78 (2009)
20. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Trans. on Software Engineering and Methodology (TOSEM) 21(3), 14:1–14:39 (2012)
21. Kramer, D., Oussena, S., Komisarczuk, P., Clark, T.: Using document-oriented guis in dynamic software product lines. ACM SIGPLAN Notices 49(3), 85–94 (2013)
22. Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C.: Scalable analysis of variable software. In: European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE). pp. 81–91. ACM (2013)
23. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55 – 92 (1991), http://www.sciencedirect.com/science/article/pii/0890540191900524
24. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 71–84. POPL '93, ACM, New York, NY, USA (1993)
25. Schnieders, A., Puhlmann, F.: Variability mechanisms in e-business process families. BIS 85, 583–601 (2006)
26. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Computing Surveys (CSUR) 47(1), 6:1–6:45 (2014)
27. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. ACM SIGPLAN Notices 48(3), 11–20 (2012)
28. Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. pp. 61–78. LFP '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/91556.91592
29. Yao, X., Shah, N.D., Sangaralingham, L.R., Gersh, B.J., Noseworthy, P.A.: Non–vitamin k antagonist oral anticoagulant dosing in patients with atrial fibrillation and renal dysfunction. Journal of the American College of Cardiology 69(23), 2779–2790 (2017)