Specifying Abstract User Interface in VDM-SL

Tomohiro Oda[1], Keijiro Araki[2], Yasuhiro Yamamoto[3], Kumiyo Nakakoji[3,1], Han-Myung Chang[4], and Peter Gorm Larsen[5]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] National Institute of Technology, Kumamoto College (araki@kyudai.jp)
[3] Future University Hakodate (yxy@acm.org, kumiyo@acm.org)
[4] Nanzan University (chang@nanzan-u.ac.jp)
[5] Aarhus University, DIGIT, Department of Engineering, (pgl@eng.au.dk)

Abstract. Interactive systems are often equipped with graphical user interfaces using complex states, constraints and computation to bridge between the rest of the system and the user. Elements on a graphical user interface are dynamically created, laid out, styled and set up by event handlers according to the internal state of the system. This paper introduces ViennaVisuals, a framework to construct XML-based abstract graphical user interface models in VDM-SL which is illustrated with an example.

1 Introduction

A Graphical User Interface (GUI) is a powerful mechanism to allow the user to interact with a software system. Objects rendered on the screen represent a part of the system's internal states, and also provide cues to trigger the next available operations whose preconditions are satisfied. In addition to navigation based on the systems functional model, a user-friendly GUI also guides the user to appropriate operations according to prospective use scenarios. Thus, GUI design should implement both the prospective use scenarios and the preconditions of operations available to the user.

User validation of the formal specification is a strength of the VDM-SL tool support. The executable subset of the VDM-SL notation allows the developers to evaluate the validity of the specified functionality using specification animation. A formal specification in VDM-SL defines each operation to the system by specifying inputs, outputs, and the change of the internal state. Every operation specified in VDM-SL has its precondition based on the internal state and the inputs, and the internal state can have an invariant assertion. One can therefore analyse the possible sequence of operations and their inputs based on the preconditions and the changes of internal states. On the other hand, VDM-SL does not have language constructs dedicated to the explicit definition of user scenarios. Although one can write a sequence of operation calls inside an operation to express one concrete user scenario, such operation calls are rarely enough to evaluate the validity in general cases.

The actions to be taken by a user and the corresponding sequence of operations that makes sense to the user often depend on the information presented to the user. The user is not simply performing a predefined sequence of operations using a GUI. The user does not operate on the GUI to only obtain or express the solution, but also gradually revises and confirms the decisions that the user is making [4].

The authors have been studying animation tools to support use scenarios with VDM-SL. The Overture tool has an interpreter console and a debugger to animate executable specification in VDM-family [2]. A GUI generator to drive a specification was also developed on the Overture tool [5]. ViennaTalk is an IDE for VDM-SL designed to support the exploratory process of the early stage of the specification phase and provides a UI prototyping tool called Lively WalkThrough and a WebAPI server called Webly WalkThrough [6]. In those animation tools, GUIs are not included in the formal specification, but they are prototyped outside the formal specification. However, considering the increasing functional complexity of GUIs for interactive systems, the benefits of specifying GUIs as a part of the formal specification is becoming significant.

This paper proposes ViennaVisuals, a framework that enables VDM-SL to specify internal states and constraints on GUI in Section 2, followed by a concrete example in Section 3. The design of ViennaVisuals is discussed in Section 4. Afterwards, Section 5 explains other researches on visual components of model-based formal specifications. Finally, Section 6 provides a few concluding remarks.

2 ViennaVisuals

VDM-SL is a formal specification language that enables mathematical analysis on computational systems. A conventional VDM-SL specification typically defines a functional model using internal states and operations that read and/or write the states. The internal states are constrained by types and invariant assertions. The operations represent functionalities that the specified system provides. An operation is typed and also its invocation is constrained by a precondition. One can analyse the mathematical property of the satisfiability of preconditions and invariants, such as deciding whether or not a particular sequence of operations are possible and safe to call at a given state. The executable subset of VDM-SL enables the developers to walk-through an expected use scenario.

The user interface is another model of the system that defines a possible sequence of operations and the validity of the system in use scenarios. A GUI is typically stateful in the sense that the graphics on the screen and the system's response to the user's manipulation changes dynamically. A GUI should satisfy both the validity in the expected scenarios and the constraints by the system's functional model. An operation that leads to violation of any of the invariants on the internal system should not be enabled on the GUI.

ViennaVisuals is a framework for VDM-SL that provides a VDM-SL module to specify XML-based GUIs and also drives the GUIs on a web browser. Specifying a GUI is not just drawing GUI widgets using a graphics library. Although an implemented system may have a GUI on a bitmap display, the output of the GUI specification is not pixels, but an abstract presentation of visual objects. The GUI represents the system's functionality as well as an interpretation of the internal state of the system. ViennaVisuals represents visual objects in an Abstract Syntax Tree (AST) of geometric shapes in a similar way that ReadEvalPrint Loop (REPL) interpreters represent the language's expressions in the ASTs.

ViennaVisuals is implemented as a component of ViennaTalk [6]. ViennaTalk has a HTTP server called Webly WalkThrough that can animate VDM specifications and a parser library to convert a VDM expression into various formats. The implementation of ViennaVisuals consists of an extension to Webly WalkThrough and a library in VDM-SL to construct ASTs of XML elements. The extension to Webly Walk-Through includes a converter from AST in VDM-SL to the XML format, a JavaScript code to send HTTP requests to the Webly WalkThrough server, and animation manager to drive the UI model specified in VDM-SL.

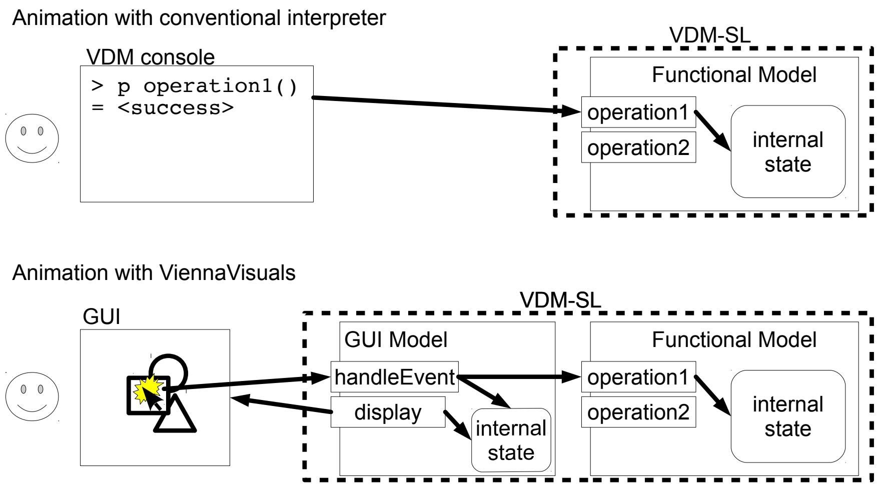

Fig. 1. Animation with the conventional VDM interpreter and ViennaVIsuals

Figure 1 illustrates the specification animation with ViennaVisuals. Assuming that the specified system has two public operations `operation1` and `operation2` and the user is validating the `operation1` in a certain scenario. The system has its internal state in the functional model. When the user animates the use of the public operation `operation1`, the user types an expression to call `operation1`. The interpreter evaluates it and shows the resulting value on the screen. Because the VDM-SL specification is the functional model of the specified system, the specification does not tell how the user gives the command to the system with the user interface. The user may not know how the result of the operation will be shown to the user.

In the animation with ViennaVisuals, the user sees graphical objects on the screen. The system has a GUI model along with the functional model. The GUI model exports one operation for an event handler, typically named `handleEvent`, and another operation to render the GUI, typically named `display` by convention. When the user clicks on a rectangle, ViennaVisuals executes the associated event handler, in this case handler1, which will call the operation1 operation of the functional model. ViennaVisuals will then execute the display operation to generate the GUI shown as the result to the user.

Both the GUI model and the functional model are written in VDM-SL. The specifier can use tools other than ViennaVisuals for further analysis, such as proof obligation generators and testing frameworks. The GUI model has two responsibilities; constructing visual objects to display and handling UI events from the user. The remainder of this section will explain them one by one.

2.1 Constructing visual objects

The construction of visual objects is one of the major features that a GUI framework should provide. ViennaVisuals provides a mechanism to synthesise and show graphics to the user. While some GUI frameworks implemented in and for programming languages provide imperative commands to draw graphics on a bitmap screen, ViennaVisuals takes an approach called DOM (Domain Object Model) tree. Each node in a DOM tree represents a visual object, and one DOM tree represents all visual objects on the screen. ViennaVisuals defines the DOM in VDM-SL and therefore tools for VDM can process visual objects as values in VDM-SL.

ViennaVisuals expects a GUI module to define an operation, typically named display, that generates a DOM tree. ViennaDOM[6] is a module that provides types and operations to generate XML-based DOM elements. Although ViennaDOM can handle arbitrary XML-based formats, ViennaVisuals is designed to use primarily the SVG (Scaled Vector Graphics) format. SVG is an XML-based format to define graphical shapes, such as rectangles, circles, lines, polygons, and texts.

Figure 2 shows the definition of the TaggedElement type that represents the AST in XML. The attributes field holds attributes each of whose values are either a string or a real number. Conversion functions such as integer to string and vice versa are also provided for convenience. The contents field holds its child elements nested in the parent element. For example, `mk_TaggedElement("A", {|->}, {}, [mk_TaggedElement("B", {|->}, {}, [], {}, 2), "C"], {}, 3)` will be rendered as the following XML extract:
`<A identifier="3"><B identifier="2"></B>C</A>`.

The tokens fields stores marker tokens so that the element can be retrieved later from the current document tree. The tokens stored in the tokens fields are not rendered in the XML document.

ViennaDOM also provides a set of operations and functions to create, modify, and search for XML elements. Figure 3 shows the state definition, major operations and functions to construct XML documents. The operations and functions shown in Figure 3 are designed to conform the DOM API in HTML 5.

The createElement operation creates an XML element tagged with the given name. For example, `createElement("rect")` returns a SVG element that will be rendered as `<rect identifier_=...></rect>`. Use of the createElement

[6] The source of ViennaDOM is available at https://github.com/overturetool/documentation/blob/editing/examples/VDMSL/xml/ViennaDOM.vdmsl.

```
types
     Element = TaggedElement| String;
     TaggedElement ::
        name : Name
        attributes : map Name to [String| real]
        eventHandlers : set of EventType
        contents : seq of Element
        tokens : set of token
        identifier_ : nat;
     String = seq of char;
     Name = seq1 of char
     inv [c]^str == c not in set IllegalNameStartChar
            and elems str inter IllegalNameChar = {};
     Point :: x : int y : int;
```

Fig. 2. The definition of `TaggedElement` and `Element` types for XML elements

```
state DOM of
    current : Element
    nextIdentifier : nat
init s == s = mk_DOM("", 0)
end

operations
    createElement : String ==> TaggedElement
    pure getElementById : String ==> [TaggedElement]
    pure getElementsByToken : token ==> seq of TaggedElement

functions
    setAttribute : TaggedElement * String *
               (String | real | seq of Point) -> TaggedElement
    getAttribute : TaggedElement * String -> [String]
    hasAttribute : TaggedElement * String -> bool
    removeAttribute : TaggedElement * String -> TaggedElement
    appendChild : TaggedElement * Element -> TaggedElement
    addToken : TaggedElement * token -> TaggedElement
    hasToken : TaggedElement * token -> bool
```

Fig. 3. The signatures of major operations and functions to manipulate XML elements

instead of the record constructor mk_TaggedElement(...) is highly recommended to properly manage the identifier_ field. The TaggedElement type has the identifier_ field to trace identities of XML elements between VDM-SL and Java-Script. The state variable nextIdentifier manages to provide values for the identifier_ field. The invocation of createElement operation thus makes a change of state and is defined as an impure operation.

The getElementById operation retrieves an XML element from the current document tree. The getElementsByToken operation retrieves all the XML elements that hold the marker token given as the argument.

The setAttribute function takes the XML element to manipulate, the attribute name, and its value to set, and returns the resulting XML element. The reason why it is defined as a function is that VDM-SL does not provide mutable objects. The DOM API of HTML 5 provides XML elements as mutable objects and their methods naturally modify its content. The values of the TaggedElement type are immutable as well as any other values in VDM-SL. The only state variables are mutable in VDM-SL. Although it is technically possible to define an operation that modifies the current document tree by replacing it with a new XML element, the performance will be quite inefficient. While programs in HTML 5 often modify the DOM tree incrementally, the use of the document tree in ViennaVisuals would tend to be constructive than mutative. We therefore designed setAttribute as a function. The functions getAttribute, hasAttribute, appendChild are defined in a similar manner.

The functions addToken and hasToken are used to set and test marker tokens. Although marker values could be set into attributes, the type of the attribute values is limited to String in XML. By giving a value of the **token** type, an arbitrary value in VDM-SL could be used as a marker token to trace particular XML elements.

2.2 Event handling

This section describes the event handling with ViennaVisuals. The event handler is a simple mechanism used in various GUI platforms. When the UI framework detects that the user made an action at a visual object, the event handler is invoked to process the action. While many UI frameworks associates event handlers with each visual object, ViennaVisuals expects the GUI module to define one operation, typically named handleEvent, that processes events triggered in the GUI.

Figure 4 shows the type definitions and the signatures of major functions defined in the ViennaDOM module. ViennaVisuals currently supports two types of events; the <click> event and the change event. To receive an event triggered at a visual object, the DOM element of the visual object should be registered to handle events.

The registration is carried out in the DOM tree. The setEventHandler function is an auxiliary function to set the registration on the DOM element given as the first argument and the type of the event specified as the second argument. The event handler fields of DOM elements are rendered as onclick or onchange attributes in the generated XML document. An event handler operation should be implemented with the signature Event ==> (). The argument could be either a mouse event or a change event. Either event type has the type field and the target field. The target field stores the XML element where the event occurred.

```
types
     EventType = <click> | <change>;
     Event = MouseEvent| ChangeEvent;
     MouseEvent ::
      type : EventType
      target : TaggedElement
      x : nat
      y : nat;
     ChangeEvent ::
      type : EventType
      target : TaggedElement
      value : String;

functions
    setEventHandler : TaggedElement * EventType -> TaggedElement
    pure getElement : nat ==> [TaggedElement]
```

Fig. 4. Type definitions and major functions of the `ViennaDOM` module

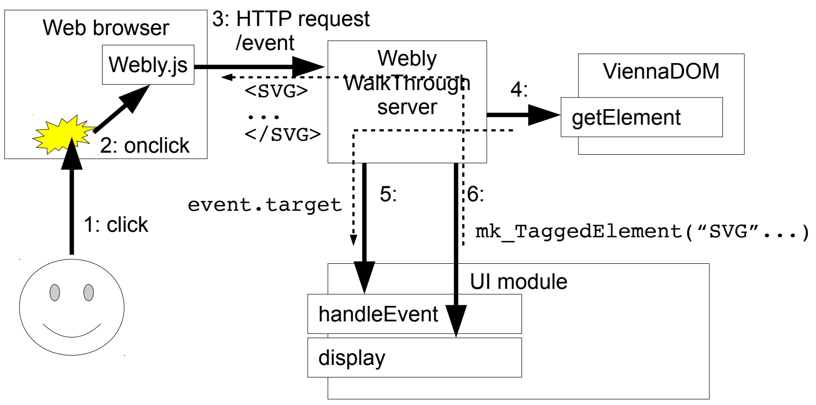

Fig. 5. Event handling in ViennaVisuals

The mechanism to handle a GUI event is illustrated in Figure 5. The thick arrows with numbers show the sequence of messages. The thin dashed arrows indicate the flow of data passed along the messages.

A click at a visual object on the browser (the message 1 in the figure) invokes a function in `Webly.js` (message 2). The function then sends an HTTP request to the Webly WalkThrough server on ViennaVisuals (message 3). The `identifier` attribute of the visual object is passed in the request. The Webly WalkThrough server has to invoke the `handleEvent` operation and gives an event value as an argument to process the event.

Because ViennaVisuals use web browsers as the client platform and VDM-SL as the specification language, a visual object exists both in the web browser and inside the VDM-SL model. To have the DOM element of the event as the parameter, traceability from the browser's DOM element to the ViennaDOM's element is crucial. The `getElement` operation retrieves the DOM element with the specified `identifier_` field from the `current` DOM tree (message 4). An event value is composed using the information given by the HTTP request and the return value of the `getElement` operation. The `handleEvent` operation is called with the event value as the argument (message 5). The event has been processed by this call.

The event handler defined in the GUI module may typically change the internal state of the system. The visual objects on the GUI need to be updated. ViennaVisuals calls the `display` operation to obtain the updated DOM tree (message 6). The Webly WalkThrough server then converts the DOM tree value into the XML format and sends it to the browser as the response to the HTTP request. The browser will render the SVG document and the user is ready to make a new action.

Both the `display` operation and the `handleEvent` operation are defined in VDM-SL. They are just conventional operation with no binding with external native libraries so that interpreters and analysis tools for VDM-SL can process the operations. For example, one can write a unit test to assert the result of operation calls on the `handleEvent` performs an expected side-effect.

3 Example: Reviewers' bidding

This section introduces an example application of ViennaVisuals to explain how a dynamic GUI can be specified based on a functional model. The application is a bidding system for peer-reviewing.

Peer reviewing is widely practiced in academic publications such as proceedings and journals. Bidding is one method to assign papers to each reviewer. A reviewer chooses a certain number of papers of which the reviewer is confident of expertise for judging the quality. The choice is called a *bid*. Reviewers will be then selected for each paper based on the bids from all reviewers.

On the other hand, there are rules that constrain who can review which papers. In this section, we will respect the following three rules.

1. A reviewer can make three bids.
2. A reviewer cannot bid on the papers authored by the reviewer.

3. A reviewer can bid on at most two papers authored by the same person.

Rule 1 defines the number of papers that a reviewer can bid. Rule 2 prohibits self-reviews while rule 3 is to avoid situations that one reviewer has too much influence on one particular author's work. It is easy to write the three rules in VDM-SL, and apply the constraint on the "bid" operation as the precondition. Rule 3 defined in VDM-SL is shown in Figure 6. The validation function `isValidBid` can be defined using such functions as shown in Figure 7, and is used as precondition of the `bid` operation. The `cancelBid` operation is also defined. The operations `numPapers`, `getPaper` and `getBids` are also defined similarly. These functions and operations are core parts of the functional model of the bidding system. They can be tested using unit testing and combinatorial testing to check their mathematical correctness.

```
functions
  exceeds_max_papers_from_same_author : set of Paper -> bool
  exceeds_max_papers_from_same_author(papers) ==
     exists author in set dunion {elems paper.authors
                                 | paper in set papers} &
            card {paper | paper in set papers
                        & author in set elems paper.authors}
            > MAX_PAPERS_FROM_AUTHOR;
```

Fig. 6. An example definition of bidding rules: number of papers from the same author

```
functions
  isValidBid : Person * set of Paper -> bool
  isValidBid(reviewer, papers) ==
      not ((has_conflict_of_interest(reviewer, papers)
              or exceeds_max_bids_per_reviewer(papers))
          or exceeds_max_papers_from_same_author(papers));

operations
   bid : Person * Paper ==> ()
   bid(person, paper) ==
       bids := bids union {mk_(person, paper)}
   pre canBid(person, paper);
```

Fig. 7. The definition of validation function for bidding and the `bid` operation

However, the mathematical correctness of the definitions in the functional model is just one of the two sides of the coin. We also need to check whether or not the set of public operations `numPapers`, `getPaper`, `getBids`, `canBid`, `isBidded`, `bid`, `cancelBid` is feasible for the user to make appropriate bids. The three rules of bidding are not complex, but still require a cognitive burden for the user to operate correctly.

One important responsibility of GUI is to guide the user to operate the system. The operational feasibility of a functional model partly depends on the GUI. One advantage of GUI is expressiveness; the GUI can tell the user which paper is legal to bid, which paper is no longer legal to bid, and which paper has already received bids.

```
state BiddingUIState of
    reviewer : Person
init s == s = mk_BiddingUIState("Alice")
end
```

Fig. 8. State definition of `BiddingUI` module

A GUI is used by one user at a time. The GUI module named `BiddingUI` has the `reviewer` variable to capture who is the user as shown in Figure 8. The `display` operation defined in Figure 9 renders a list of the submitted papers. Each paper is displayed as a text enclosed by a rounded rectangle. A token with the paper index is set to the rectangle element so that the event handler can later retrieve the paper from the visual element. If a paper is legal to bid or has already been bid for, the rounded rectangle is set clickable. Otherwise, the paper is illegal to bid and thus the text is in `silver` to indicate it is disabled. The already bidded papers are also marked by a check sign. These are the mapping rule from each paper to its visual presentation supplemented with the contextual information by the bidding rules. With the visual objects, the user's task is now reduced to look for a rounded rectangle without a check sign and click on the paper with the most confident of refereeing.

The event handler is defined in Figure 10. The handler accepts any event but responds only to `<click>` events. The handler then toggles the bidding state of the paper using the token attached to the visual element stored in the `target` field.

Figure 11 illustrate the flow of making bids. A reviewer *Alice* opened the UI to make bids. At the step 1, the system shows the list of submitted papers and two papers (Paper1 and Paper3) are disabled because `Alice` is a co-author. Alice clicks on Paper4 written by Dan, Eike and Fox. The system then put a check sign at Paper4 as shown at the step 2. Paper2, Paper5, Paper6 and Paper7 are available and Alice choose Paper6. The system again put a check sign at Paper6 and disables Paper7 at the step 3 because bidding on Paper7 would violate the bidding rule 3. Fox is a co-author of Paper4 and Paper6 and Paper7 is also co-authored by Fox. Bidding rule 3 prohibits bidding more than two papers authored by the same person.

```
operations
  display : () ==> TaggedElement
  display() ==
    (dcl list:TaggedElement := createElement("svg");
     for index = 1 to  numPapers()
     do
        let paper : Paper = getPaper(index)
        in
           (dcl itemText:TaggedElement,
                itemRect:TaggedElement;
           itemRect := frame(index);
           itemText := title(index);
           itemRect := addToken(itemRect, mk_token(index));
           if canBid(reviewer, paper)
              or isBidded(reviewer, paper)
           then itemRect :=
                   setEventHandler(itemRect, <click>)
           else itemText :=
                    setAttribute(itemText, "fill", "silver");
           list := appendChild(list, itemRect);
           if isBidded(reviewer, paper)
           then list := appendChild(list, check(index));
           list := appendChild(list, itemText));
    return list);
```

Fig. 9. The display operation of `BiddingUI`

```
operations
  handleEvent : Event ==> ()
  handleEvent(event) ==
      cases event:
          mk_MouseEvent(<click>, target, x, y) ->
              let index in set {1, ..., numPapers()}
              be st hasToken(target, mk_token(index))
              in toggleBid(getPaper(index)),
          others -> skip
      end;
```

Fig. 10. The handleEvent operation of `BiddingUI`

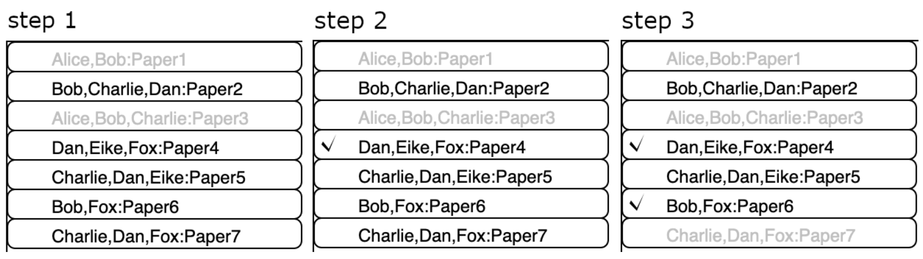

Fig. 11. Steps in bidding for a reviewer *Alice* with the GUI

The specifier can walk through the scenario described above on a web browser. Also the scenario above can also be written as a unit test using operation calls to the `display` operation and the `handleEvent` operation, and run on a CI server.

4 Discussions

The major features in the design of ViennaVisuals are:

– XML as the representation of visual objects,
– one `display` operation to generate an AST of XML, and
– one `handleEvent` operation to process UI events.

The objective of ViennaVisuals is not to draw fancy graphics, but to *specify* the UI of the system: how the system should guide the user by visual objects, and how the system should respond to the user's actions. This section discusses the design choices of ViennaVisuals in the perspective of engineering tasks in formal modeling.

4.1 Formal abstraction of GUI

ViennaVisuals models the GUI as visual objects composed of geometric shapes and discrete events triggered by the user's manipulations. Comparing to the imperative graphics library with commands such as drawLine, drawCircle, and drawRectangle, ViennaVisuals can handle the result of the rendering as the first-class object. The `rect` element is a visual object that can be passed to a function, stored into a state variable, and referred to in the proof obligations.

The events are defined as records in VDM-SL. ViennaVisuals generates an event value to be passed to the `handleEvent` operation. Although the generation of the event value is done by the runtime of ViennaVisuals, the event value is a first-class object and subject to formal analysis.

4.2 Centralised specification of event handling

ViennaVisuals assumes one `handleEvent` operation in a GUI module. Many GUI frameworks allow each visual object to have their own event handlers while ViennaVisuals expects the only one operation per GUI model. Because operations are not first-class objects in VDM-SL, an operation cannot be passed as an argument or be assigned to a field of a record value. Although it is possible to design a GUI framework to register the name of the event handler operation as a string, it would make it difficult to analyse why an event handler is dispatched by a UI event because traceability from the string to the event handler operation would be missed. Thus passing the event handler name as a string could be problematic in the formal analysis.

With the current design of ViennaVisuals, the event handler operation can take the benefit of the strong pattern matching mechanism of VDM-SL as seen in Figure 10. Without having each visual object with its own event handler, VDM-SL can concisely abstract the event handling process in its own way.

4.3 Time granularity of interactions

The current implementation of ViennaVisuals uses web browsers as the UI platform and HTTP as the protocol between VDM-SL and the selected web browser. The GUI is not just displaying the result of the computation but provides communication between the user and the system. Some GUIs use visual effects, such as little vibration of the visual object, to enable subtle interactions between the user and the visual object. Such visual effects typically happen in the time granularity of milliseconds. ViennaVisuals is not good for specifying such effects because of the overhead by HTTP and data migrations between JavaScript and VDM-SL. Although SVG supports some of such visual effects, the effects are not the first-class objects in VDM-SL and thus out of the scope of formal analyses.

ViennaVisuals is designed to specify interactions in the order of seconds. It should be also noted that VDM-SL does not support temporal features such as the **`time`** variable in VDM-RT. ViennaVisuals is not appropriate for specifying the precise timing of the interactions.

4.4 Extensibility to other XML-based UIs

ViennaVisuals provides the DOM construction library named `ViennaDOM`, which can handle general XML elements. Although the design of ViennaVisuals aims at SVG as the document format, it is possible to specify the `display` operation to generate other XML-based models, such as VoiceXML[7] for Voice UI and X3D[8] for 3D graphics.

4.5 Limitations

The `display` operation is impure because the `createElement` operation in the `ViennaDOM` module is impure, which could be obstacle to formal analysis. The

[7] `https://www.w3.org/TR/voicexml21/`
[8] `https://www.web3d.org/x3d/what-x3d`

`createElement` operation needs to be impure to manage the traceability between the visual elements in XML and in VDM-SL. The traceability is mandatory to retrieve the target of a UI event when invoking the `handleEvent` operation. The traceability could be managed within the `display` operation, but a failure of managing the traceability could result in destroying the semantics of the event mechanism.

The design of ViennaVisuals does not fit with multi-threading. ViennaVisuals assumes that the internal state is kept after the last display() call until the next event callback invokes the `handleEvent` operation. If the state is changed between the `display` operation and the `handleEvent` operation, the traceability will be broken and thus the semantics of the event mechanism will be unreliable. VDM-SL does not support multi-threading, but this is a limitation of the design of ViennaVisuals when applying to other formalisms.

5 Related Work

5.1 PVSio-web

PVSio-web [3] is a graphical toolkit to platform to evaluate a formal model defined in PVS and user interface. It is used in practice to evaluate safety of medical devices with combinations of user interface and a formal model. PVS is a formal modeling system based on theorem provers and animation. PVSio-web drives realistic user interfaces so that operations using the interface can be carried out safely. ViennaVisuals does not aim at operating a realistic GUI but the abstract specification of the user interface based on visual shapes and event handling.

5.2 Lively WalkThrough

Lively WalkThrough [6] is another GUI prototyping framework in ViennaTalk, which aims at encouraging communication between the formal specification engineers and UI designers. In Lively WalkThrough, one can compose a UI prototype on a given VDM-SL specification and define event handlers for each visual component in the UI prototype. By walking through a user scenario on the UI prototype, the formal specification engineer can see how the system's functionality will be used and the UI designer can see how the user scenario can be carried out by the system's functionalities. Because the focus of Lively WalkThrough is to drive the specification with a prototypical UI, the UI and event handling is not specified in VDM-SL but separately defined on the tool.

5.3 Crescendo/Symphony/INTO-CPS

Crescendo, Symphony, and INTO-CPS are co-simulation tools based on the Overture tool [1]. The three co-simulation tools drive two different types of simulations together: functional models with discrete events defined in VDM, and physical models with continuous-time using simulation engines. Some components of the co-simulation tools provide 3D graphics to visualise the virtual physical model in the simulation model. Although the co-simulation does not aim at GUI design, the visualisation can be seen as a graphical presentation of the internal state of the co-simulated models.

6 Concluding Remarks

A specification of the GUI is the specification of what the user will see through the system. A graphical user interface is the meeting point between the user and the system. The validity of the GUI should take both the views from the user and also from the system into the account. The GUI from the user's side of the view possibly involves human cognition. This side of the GUI is hard to be subject to formal modeling. The other side of the GUI is the computational system. Formalisms including VDM are designed to capture this side.

ViennaVisuals is a step to combine the two sides. A GUI model defined with ViennaVisuals can be analysed together with the functional model specified in the same formal specification language. The GUI model does not represent the user's cognition and emotions, but it could hopefully let the user virtually preview how the realised system would look in situations.

Acknowledgments

The authors thank anonymous reviewers for their thoughtful and constructive feedback. A part of this research was supported by JSPS KAKENHI Grant Number JP 18K18033 and 19K11911. We would also like to express our thanks to the anonymous reviewers.

References

1. Foldager, F., Larsen, P.G., Green, O.: Development of a Driverless Lawn Mower using Co-Simulation. In: 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems. Trento, Italy (September 2017)
2. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
3. Masci, P., Oladimeji, P., Zhang, Y., Jones, P., Curzon, P., Thimbleby, H.: PVSio-web 2.0: Joining PVS to HCI. In: Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 470–478 (2015)
4. Nakakoji, K., Yamamoto, Y.: chap. Conjectures on How Designers Interact with Representations in the Early Stages of Software Design, pp. 381–400. Chapman & Hall (October 2013)
5. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
6. Oda, T., Araki, K.: ViennaTalk: An Integrated Specification Environment Focused on the Early Stage of the Formal Specification Phase. Computer Software 34(4), 4_129–4_143 (November 2017), https://www.jstage.jst.go.jp/article/jssst/34/4/34_4_129/_article/-char/ja/