Proceedings of the 18th International Overture Workshop

7th December 2020

John S Fitzgerald and Tomohiro Oda (Eds.)

Preface

The 18th in the "Overture" series of workshops on the Vienna Development Method (VDM), its associated tools and applications, was held online, on 7th December 2020. VDM is one of the longest established formal methods, and yet has a lively community of researchers and practitioners in academia and industry grown around the modelling languages (VDM-SL, VDM++, VDM-RT) and tools (VDMTools, VDMJ, ViennaTalk, Overture, Crescendo, Symphony, and the INTO-CPS chain). Together, these provide a platform for work on modelling and analysis technology that includes static and dynamic analysis, test generation, execution support, and model checking.

Research in VDM and Overture is driven by the need to develop industry practice as well as fundamental theory. The 18th Workshop reflected the breadth and depth of work supporting and applying VDM. In this technical report, we include two technical sessions with papers. The first session consists of three papers associated with tool building and cloud computing systems, and the second consists of two papers on co-simulation and Cyber-Physical Systems.

As a consequence of the pandemic that gripped the world in 2020, this was the first Overture Workshop to be conducted entirely online. Thinking back over the 17 previous times on which we had shared our meetings, as well as hospitality, in person in locations around the world, this was a bittersweet event. It was bitter in the sense that we were separated from colleagues and friends, and that pressures of other tasks had kept the contributions to a minimum. At the same time, it was a great pleasure to share a few hours again with colleagues and friends with whom we have, over decades, come to share scientific interests and enthusiasms.

We would like to thank the authors, PC members, reviewers and participants for their help in making this a valuable and successful workshop, and we look forward together to meeting once more in a safer world in 2021.

John Fitzgerald, Newcastle
Tomohiro Oda, Tokyo

Programme Committee

Keijiro Araki, National Institute of Technology, Kumamoto College, Japan

Nick Battle, Newcastle University, UK

Luis D. Couto, Forcepoint, Ireland

Fuyuki Ishikawa, National Institute of Informatics, Japan

Hugo Daniel Macedo, Aarhus University, Denmark

Paolo Masci, National Institute of Aerospace (NIA), USA

Ken Pierce, Newcastle University, UK

Marcel Verhoef, European Space Agency, The Netherlands

Contents

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.
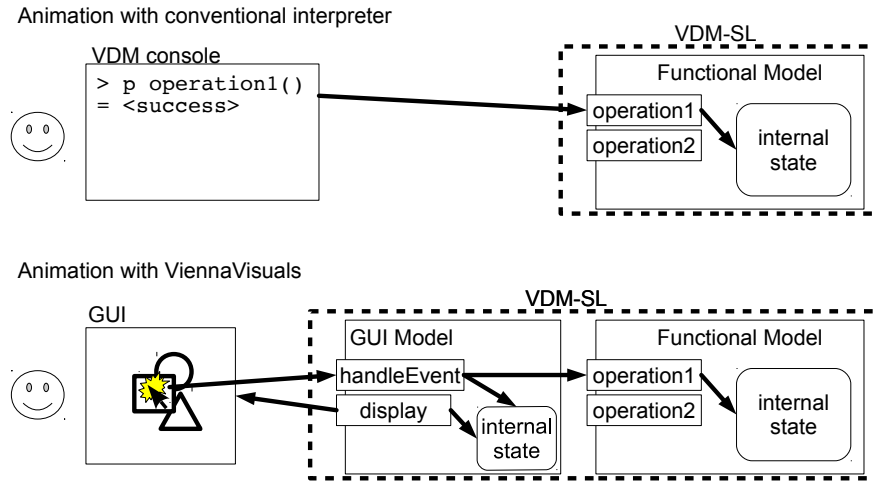
2   ViennaVisuals

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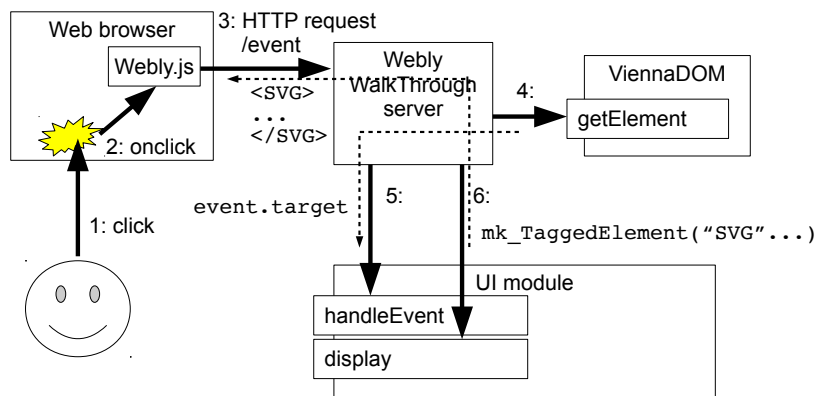
```
types
    EventType = <click> | <change>;
    Event = MouseEvent| ChangeEvent;
    MouseEvent ::
     type : EventType
     target : TaggedElement
     x : nat
     y : nat;
    ChangeEvent ::
     type : EventType
     target : TaggedElement
     value : String;

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`

step 1	step 2	step 3
Alice,Bob:Paper1	Alice,Bob:Paper1	Alice,Bob:Paper1
Bob,Charlie,Dan:Paper2	Bob,Charlie,Dan:Paper2	Bob,Charlie,Dan:Paper2
Alice,Bob,Charlie:Paper3	Alice,Bob,Charlie:Paper3	Alice,Bob,Charlie:Paper3
Dan,Eike,Fox:Paper4	✓ Dan,Eike,Fox:Paper4	✓ Dan,Eike,Fox:Paper4
Charlie,Dan,Eike:Paper5	Charlie,Dan,Eike:Paper5	Charlie,Dan,Eike:Paper5
Bob,Fox:Paper6	Bob,Fox:Paper6	✓ Bob,Fox:Paper6
Charlie,Dan,Fox:Paper7	Charlie,Dan,Fox:Paper7	Charlie,Dan,Fox:Paper7

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

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/`

Modelling the HUBCAP Sandbox Architecture In VDM: A Study In Security

Tomas Kulik[1], Hugo Daniel Macedo[1], Prasad Talasila[1], and Peter Gorm Larsen[1]

DIGIT, Department of Engineering, Aarhus University

Abstract. In this paper, we report on the work in progress towards the security analysis of a cloud based collaboration platform proposing a novel sandboxing concept. To overcome the intrinsic complexities of a full security analysis, we follow the formal methods approach and used separate teams: one developing a model and one implementing the platform. The goal is to create an abstract formal model of the system, focusing on the critical dataflows and security pain points, which create results to be shared among teams. The work is in progress, and, in this paper, we provide insights on a first model covering: the main features of the platform's sandboxing concept, and an analysis on the table of roles and profiles by the users, that regulates the user access to the different sandboxing capabilities. The model is specified in VDM-SL and was validated using the Overture tool combinatorial testing. The analysis has uncovered some potential issues, that have been shared with the platform implementation team, who have used it to more clearly define the table of roles and profiles within the system. Although useful, the work on implementing and securing the sandboxing is still in its initial stages, and given the novelty of the sandboxing concept, a definite answer regarding its security aspects is unclear, but a challenging research question.

Keywords: Sandbox · Collaborative Platforms · Model Based Engineering.

1 Introduction

A new collaborative platform for model based development (MBD) is emerging from the HUBCAP project [11, 12]. Differently from the traditional collaborative platforms, where users are given access to shared documents or models (with an interface to perform changes), the HUBCAP project provides multiple virtual machines (VMs) that are fully accessible via a browser web page. Users are given access to read and write to the operating system's file systems and can jointly edit the state of the sandbox components. Whether the concept will be widely adopted, or it suits only the needs of the MBD community is uncertain. What is known is that there is a need to perform a thorough analysis of the security implications of the design.

Conceptually, a HUBCAP sandbox is a group of virtual machines managed and hosted by a trusted third-party cloud provider. The users of a sandbox are MBD providers or adopters, which despite not being granted access to the source code of the platform details, receive VMs prepared with all the required dependencies to run and develop tools and models. The advantages of the concept are twofold. Firstly, the users can explore and interact with a system without having to give permission to either their local

machine OS facilities or data. This goes beyond remote assistance tools that allow an external party to provide training or help configure the requesting users' local machine. Secondly, the users are able to manipulate a whole system, beyond a file or a view of the model data, which is interesting when the collaboration involves the development of cyber-physical systems and its associated need to use multiple tools and configure various system parameters. On the down side, and beyond the obvious resource consumption, the concept is new and the security requires trust between all the users and the cloud third party provider.

This paper is the first step into a full analysis of the security aspects of the HUBCAP platform. The work lays the foundation for future extensions to the Sandbox concept and evaluates whether it is possible to ensure security in multiple scenarios. For example, if the third party cloud provider is compromised, or proven to be untrustworthy.

To study the security properties of the HUBCAP platform we are developing a Vienna Development Method (VDM) model of the system components and functionality that is security critical. VDM is a formal method prescribing the development of models of computer systems, which are used to generate code for the system implementation, or to provide insight to the implementation team. The models have a precise semantics and have proven to be useful in the process of software development and security analysis.

Within this analysis we have used VDM-SL a specification language standardised by the International Standardisation Organisation [7, 15]. It is a model-oriented formal specification language which is based around logic (e.g., for invariants, pre-conditions and post-conditions) and a number of abstract data types for different kinds of collections. It has been used industrially since the seventies where it was invented at IBM's laboratories in Vienna [10, 6]. For the last two decades the Overture/VDM tool support has evolved after the VDMTools area [8, 9].

Our work provided feedback to the implementation team, and led to the refinement of the platform user roles and profiles tables, which functions as an "access-control list" to the features available to the different users. Although we are far from being able to fully assert the security properties of the platform, our work was able to elicit the dataflow and pinpoint several security checkpoints that are necessary to ensure the security of the platform. We further intend to iterate our model continuously to align with potential updates to the roles and profiles table.

The rest of the paper is organized as follows, Section 2 introduces the concepts of sandboxing and how it is utilised within the HUBCAP platform. It further introduces a table of profiles and roles governing access of clients to specific functionality within the platform. Section 3 describes the formal model of the sandboxing platform created in VDM-SL with focus on the functionality of creation of sandboxes and features present in the table of profiles and roles. Section 4 provides an overview of the formal analysis carried out to ensure that the platform is secure based on the table of profiles and roles. Section 5 then defines related work in the area of using formal methods in security and sandboxing assurance. Finally Section 6 provides concluding remarks and introduces expected future work.
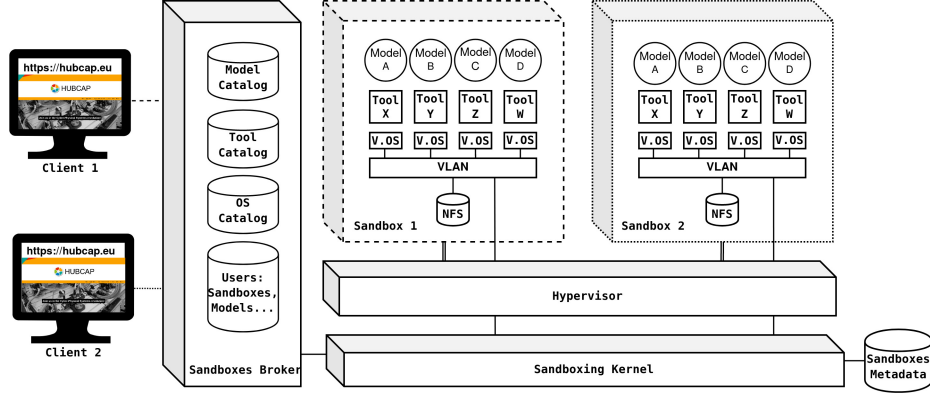
Fig. 1. The HUBCAP Sandbox architecture (taken from [11])

2   Sandboxing Platform For Collaboration

A sandbox is implemented as an isolated set of VMs (each one running a CPS tool) that interact with each other sharing a virtual dedicated subnet and a dedicated (Network File System) NFS storage service. No interaction is permitted between the VMs belonging to different sandboxes. The sandbox capability integrated with the web-platform is therefore a sort of private cloud service provider plus the middleware to manage and mediate the access to those cloud services. In addition, as many cloud service providers offer the capability to select a combination of hardware and operating systems, the HUBCAP Platform offers you to select a combination of OS environments, tools, and models to run an experiment using the HUBCAP sandbox feature.

The sandbox service is outlined in Figure 1. The web-platform is enhanced with a broker component (labelled as *Sandboxes Broker* in the figure), which hosts a web application mediating the access of different users (*Client 1* and *Client 2*) to the sandboxes they requested (*Sandbox 1* and *Sandbox 2* respectively). All the users will use an Internet browser to access the tools in the sandbox and all the interactions are mediated by the broker.

The *Sandbox Broker* has access to the catalogues of different models, tools, and preconfigured OSs that are available, so an end user can simply pick a valid combination to request a sandbox. In addition to those catalogues, the *Sandbox Broker* keeps user information, such as the user's models (private copies of the model in the catalogue, which may have been modified by the user while using the sandbox) and the sandboxes the user created. This information is important to allow the creation of new sandboxes.

The operation of user requests and the sandboxing logic is provided by the *Sandboxing Kernel*, which is a component that interacts with the system *Hypervisor* to launch the different constituents of a sandbox, namely:

– *NFS* - Network File System providing storage in the form of shared folders where model files and tool outputs are placed.
– *VLANs* - Virtual networks restricting the communications of the VMs inside a sandbox to the set of VMs composing it and those only.

– *VOS* - Virtual machines running the OSs supporting a tool, a remote desktop protocol to provide the clients access to the tool display, and other monitor and interoperability tools to operate the VM inside the Kernel.
– *Tools* - The tools running a model or a multi-model.
– *Models* - A mathematical/formal description of a component.

The operation relies on a database of metadata about the different sandboxes (the *Sandboxes Metadata* component in the figure). This component stores and keeps track of the sandboxes' states (running, suspended, . . .) and user ownership of the resources. It is worth highlighting that the Kernel has direct network connections to the Sandboxes' VLANs. We base our model on the building blocks of the sandboxing platform presented in this section, focusing on the components necessary for analysis of the client access based on roles and profiles, while abstracting away details such as VLAN addresses, specifics of the operating systems, file storage and tool properties.

2.1   Designated Roles and Activities

The sandbox platform as presented within this paper offers several features that are accessible by use of clients. Each feature is protected by requiring the client being of a specific role and profile. The full overview of features and their associated roles and profiles is shown in Table 1. The profiles within the system are provider and consumer, where the provider can add new content to the sandboxing platform such as operating systems or tools, while the consumer could utilise the content already present within the platform. Furthermore the system defines two roles on the sandboxes themselves, where the owner role is assigned to a client creating the sandbox, while a guest role is assigned to clients that have been invited by the owner to the sandbox. Without the invitation the guest cannot access the sandbox.

Table 1. Overview of profiles and roles

Feature	Provider Owner	Provider Guest	Consumer Owner	Consumer Guest
Access to remote viewer	X	X	X	X
Upload archive	X	X	X	X
Download archive	X		X	
Invite guests	X		X	
Destroy sandbox	X		X	
Select tool	X		X	
Select model	X		X	
Select operating system	X			
Save tool	X			
Upload new model	X			
Delete repository tool	X			

3 Formal Model

The model consist of several sandbox building blocks, namely (1) Client, (2) Broker, (3) Gateway, (4) Sandbox, (5) Server and (6) System. Each of these components, excluding the gateway is represented as a VDM-SL file with varying level of detail. The model considers a single default module tracking the state of all of the components. The primary purpose of the model is to determine if the system functionality behaves according to the provided table of roles and profiles. This section provides detailed overview of the different model components.

3.1 Client

The client is the entry point into the sandbox. It is a component used by the user to access and interact with the system. The system can have multiple clients, where each client has an explicit identity, expressed as `ClientId` = **nat**. Furthermore the clients that can access the system must be considered valid, this is handled by ensuring that the client identity is present in a database of valid identities stored within the system. We express this check as a pre condition on all of the client calls as **pre** `cId` **in set** `validClients;`. The client can call several operations against the broker, namely starting a new sandbox, destroying a sandbox, selecting a model from a repository, selecting a tool from a repository, selecting an operating system from a repository, inviting a guest to a sandbox, removing own repository item, connecting to a sandbox and disconnecting from a sandbox. In order to create new sandbox the client needs to first select the tool, the operating system and optionally a model that shall be present within this sandbox. The selections of the client are recorded within a `ClientSt` record and are presented together with launching of new sandbox in Listing 1.1. Once the sandbox is launched the client can access the sandbox as presented in Listing 1.2. The actual deployment of the sandbox and establishment of connections is handled by different components, namely, the broker and the system.

```
SelectOS: ClientId * OSId ==> ()
SelectOS(cId, osId)==
  clientst.selectedOS := SelectOperatingSystem(osId, cId)
pre cId in set validClients;

SelectToolFromRepository: ClientId * ToolId ==> ()
SelectToolFromRepository(cId, tId) ==
  clientst.selectedTool := SelectTool(tId, cId)
pre cId in set validClients;

SelectModelFromRepository: ClientId * ModelId ==> ()
SelectModelFromRepository(cId, mId) ==
  clientst.selectedModel := SelectModel(mId, cId)
pre cId in set validClients;

LaunchNewSandbox: ClientId ==> ()
LaunchNewSandbox(cId) ==
```

```
  StartNewSandbox(cId, clientst.selectedTool,
     clientst.selectedModel, clientst.selectedOS)
pre cId in set validClients;
```

Listing 1.1. Sandbox elements selection and sandbox launch

```
-- Connect to a sandbox
AccessSandbox: ClientId * SandboxId ==> ()
AccessSandbox(cId, sId) ==
  if not BrokerInitiateSandboxAccess(cId, sId)
  then GetError(cId, mk_token(1))
pre cId in set validClients;
```

Listing 1.2. Client accessing a sandbox

3.2   Broker

The broker is a component of the overall system that the client interacts with, before being handed over a direct connection to a sandbox. This component is aware of currently active sandboxes, their respective ownership and records the items available within the repository, specifically the operating systems, tools and models. The broker also records ownership of these repository items. This is represented by mappings as for example ownership of a sandbox recorded in `Owners = map ClientId to set of SandboxId`, while the ownership of tools is recorded as `ToolOwners = map ClientId to set of ToolId`. Furthermore the broker records how specific tools, models and operating systems are utilised within different sandboxes, an example of what is a mapping `SandboxTools = map SandboxId to set of ToolId`. A client selecting for example an operating system calls a broker operation as shown in Listing 1.3, where the precondition states that the client either needs to have a profile of a provider or a consumer as presented in Table 1.

```
SelectOperatingSystem: OSId * ClientId ==> [nat]
SelectOperatingSystem(osId, cId) ==
  return if osId in set brokerst.validOSs
        then osId
        else nil
pre cId in set brokerst.providers or
    cId in set brokerst.consumers;
```

Listing 1.3. Broker providing an operating system

The broker is also responsible for starting a new sandbox. This operation registers the sandbox within the system and it becomes available for connections. This operation further registers the requested operating system, tool and optionally a model within the sandbox and is presented in Listing 1.4. The precondition checks that the user (via the

client) has made valid selections, while the post-condition ensures that the sandbox is registered correctly with a newly generated sandbox identity.

```
StartNewSandbox : ClientId * ToolId * [ModelId] * OSId ==> ()
StartNewSandbox(cId, tId, mId, oId) ==
let sId = GenerateNewSandboxId()
in
  (brokerst.sandboxTools := brokerst.sandboxTools munion
                               {sId |-> {tId}};
   brokerst.sandboxOSs := brokerst.sandboxOSs munion
                            {sId |-> {oId}};
   if mId <> nil
   then brokerst.sandboxModels := brokerst.sandboxModels munion
                                    {sId |-> {mId}};
   systemSandboxes := systemSandboxes munion
                        {sId |-> mk_Sandbox(sId, {1})};
   if cId in set dom brokerst.owners
   then brokerst.owners(cId) := brokerst.owners(cId) union {sId}
   else brokerst.owners := brokerst.owners munion
                             {cId |-> {sId}})
pre tId in set brokerst.validTools
    and oId in set brokerst.validOSs
    and mId <> nil => mId in set brokerst.validModels
post card dom systemSandboxes = card dom systemSandboxes + 1
    and cId in set dom brokerst.owners
    and Max(systemSandboxes) in set brokerst.owners(cId);
```

Listing 1.4. Broker starting a new sandbox within the system

In order to provide connectivity to the sandbox the broker checks if the client is associated with the requested sandbox. This means that the client either needs to be an owner of the sandbox or a guest invited for access to this sandbox. This operation is shown in Listing 1.5. The precondition checks that the client actually has some role within the system.

```
BrokerInitiateSandboxAccess: ClientId * SandboxId ==> bool
BrokerInitiateSandboxAccess(cId, sId) ==
  let sandboxes = GetSystemSandboxes(),
      servers = dunion {s.sandboxServers
                        | s in set rng sandboxes
                        & s.sandboxId = sId}
  in
    (for all x in set servers do
       UpdateConnections(cId, x, sId, true);
     return true)
pre not ClientIsNull(cId,brokerst.providers,brokerst.consumers,
                     brokerst.owners, brokerst.guests) and
    ((cId in set dom brokerst.owners and
     sId in set brokerst.owners(cId)) or
    (cId in set dom brokerst.guests and
```

```
      sId in set brokerst.guests(cId)));
```

Listing 1.5. Broker initiating access to a sandbox for a client

The broker further handles operations such as assigning a guest to a sandbox invited by a sandbox owner updating the repository of items, i.e. by uploading new items or removing them, based on the wishes of the client and destroying the sandbox, thus removing it from the system.

3.3 Gateway

The gateway is a connection component providing connectivity between the sandbox and the client. More specifically the gateway opens a connection to every server that exists under a sandbox and registers which client holds the open connection to this sandbox. In the current model the focus is on analysing the specific roles of the clients against the actions they can take and hence the gateway connections are simply recorded under the system state as GatewayConnections = **map** ClientId **to set of** ServerId and also GatewayConnectionsSandbox = **map** ClientId **to set of** SandboxId. The client can connect to and disconnect from a sandbox- a functionality captured by an operation shown in Listing 1.6. The listing further shows the broker keeping a set of active sandboxes (i.e. sandboxes with an open connection).

```
UpdateConnections: ClientId * ServerId * SandboxId * bool ==> ()
UpdateConnections(cId, sId, sbId, connect)==
  if connect
  then (if cId in set dom gatewayConnections
        then atomic
        (gatewayConnections(cId):= gatewayConnections(cId)
                                   union {sId};
        gatewayConnectionsSandbox(cId):=
            gatewayConnectionsSandbox(cId) union {sbId};
        brokerst.activeSandboxes:= brokerst.activeSandboxes
                                   union {sbId}))
  elseif cId in set dom gatewayConnections
  then atomic
  (gatewayConnections(cId):= gatewayConnections(cId) \ {sId};
   gatewayConnectionsSandbox(cId):=
       gatewayConnectionsSandbox(cId) \ {sbId};
   brokerst.activeSandboxes:= brokerst.activeSandboxes \ {sbId})
```

Listing 1.6. Gateway recording the connections to a sandbox

3.4 Sandbox

Sandbox is a secure container encapsulating servers, tools and a model that the user decided to work with. The nature of the sandbox is to provide a demo environment to

users that want to try use of model based engineering within a hosted environment. The sandbox provides an isolation from other sandboxes and the wider system and users working with sandboxes follow functionality roles as shown in a Table 1. The user can, via clients, create new sandboxes, connect to sandboxes, disconnect from sandboxes and destroy sandboxes. If a user creates a sandbox, this user can further invite another user as a guest to the sandbox, via the identity of the client that the guest uses. Each sandbox within the system could be understood as a collection of servers with specific tools installed and each sandbox carries an unique identity. The definition of the sandbox is shown in Listing 1.7.

```
types
SandboxId = nat;
SandboxServers = set of ServerId;
Sandbox::
 sandboxId : SandboxId
 sandboxServers : SandboxServers
```

Listing 1.7. Sandbox specification

In the current iteration the model is only capable of creating sandboxes consisting of a single server, as this is sufficient for analysis of the access and functionality roles.

3.5 Server

Within the modelled system, the server is a machine running an operating system with a specific tool allowing a user to manipulate models with a possibility of a remote access. Each server has to be a part of a sandbox in order to be accessible and each server carries a unique identity as `ServerId = nat`. Within the model presented in this paper, the only considered functionality of the server is, the server being a part of a sandbox.

3.6 System

The System VDM file captures functionality of the system as a whole and facilitates operations not suitable for other components. The system for example generates new identities for new sandboxes as shown in Listing 1.8. Furthermore the system contains the state for the entire system and its components as shown in Listing 1.9. The system represents an abstract container for the system components with its own high level functionality.

```
pure Max: SystemSandboxes ==> nat
Max(ss) ==
  let max in set dom ss be st forall d in set dom ss & d <= max
  in
    return max
pre ss <> {|->};
GenerateNewSandboxId: () ==> nat
```

```
GenerateNewSandboxId() ==
  return Max(systemSandboxes) + 1;
```

Listing 1.8. Operations generating identity for new sandbox

```
state SystemSt of
  gatewayConnections : GatewayConnections
  gatewayConnectionsSandbox : GatewayConnectionsSandbox
  systemSandboxes : SystemSandboxes
  toolOwners : ToolOwners
  modelOwners : ModelOwners
  brokerst : BrokerSt
  clientst: ClientSt
  validClients : ValidClients
inv ss == dom ss.gatewayConnections =
        dom ss.gatewayConnectionsSandbox
init s == s = mk_SystemSt({|->},{|->},{|->},{|->},{|->},
    mk_BrokerSt
        ({},{},{},{},{},{},{|->},{|->},[],{|->},{|->},{|->}),
    mk_ClientSt(nil,nil,nil),{})
end
```

Listing 1.9. System state

4   Formal Analysis

This section specifies the traces that have been used to analyse several system features against the table of profiles and roles. The most important functionality analysed was the creation of new sandboxes and ensuring that uninvited clients do not have access to the sandboxes, an important aspect for the integrity of the platform.

4.1   Traces under analysis

To analyse the specific cyber security properties, we express several scenarios as traces and use the combinatorial testing feature of overture. The scenarios considered within this paper are creation of sandboxes, connection and disconnection to sandboxes and creation of sandboxes considering invitation of guests to the newly created sandboxes. These scenarios have been selected based on the communication with the implementation team in order to ensure that the table of roles and profiles provides the intended permissions in order to access the system. The analysis has uncovered a potential permissions issue within the system. In order to create a sandbox a trace as shown in Listing 1.10 could consider several scenarios, where the user with a profile consumer, a user with a profile provider (profiles as specified in Table 1) and a user without a specific profile attempt to create a new sandbox. To do this, the users select the operating

system, the tool and the model that shall be used within the sandbox (only the operating system is mandatory as the tool and the model could be uploaded later to an existing sandbox). Once the selection is complete the users simply launch a new sandbox. The initial operation calls in the listing simply sets up the environment, ensuring that the system has the clients, the operating systems, the tools and the models registered.

```
CreateSandboxMultipleClients:
SetupClients(1);
SetupClients(2);
SetupClients(3);
SetupProviders(1);
SetupConsumers(2);
SetupOSs(1);
SetupTools(1);
SetupModels(1);
let clientId in set {1, 2, 3}
in
(SelectOS(clientId, 1);
 SelectToolFromRepository(clientId, 1);
 SelectModelFromRepository(clientId, 1);
 LaunchNewSandbox(clientId));
```

Listing 1.10. A trace of different clients launching a sandbox

In order to initialize client within the system a `SetupClients` operation is called with further operations setting up the client as a consumer or a provider. This is shown in Listing 1.11.

```
SetupClients: ClientId ==>()
SetupClients(cId) == validClients:= validClients union {cId}

SetupProviders: ClientId ==> ()
SetupProviders(cId) ==
  brokerst.providers := brokerst.providers union {cId};
```

Listing 1.11. Setting up a client within the system

This scenario has uncovered an issue where a user without a specific role and without a specific profile (but still a valid user within the system) attempts to launch a sandbox. While this user is allowed to launch a sandbox, it is only the users with profiles *consumer* or a *provider* that are allowed to select an operating system. This is captured as a precondition on the `SelectOS`. Since the user without an explicit profile can not select an operating system, it is not possible for this user to create a new sandbox. This led to a suggestion that users that are created on the platform but do not have assigned access rights should not be able to create a new sandbox.

Another trace considered was the trace specifying a scenario where a user (using a client) creates a sandbox and another user (using a separate client) attempts to connect to this sandbox without first receiving an invitation to this sandbox. This trace is shown

in Listing 1.12. The analysis of this trace has uncovered that the permissions model based on roles and profiles specified in Table 1, expressed as a pre-condition for the operation `AccessSandbox` prevents this from happening, ensuring that only users with legitimate claim could access the sandbox, i.e. only the owners or invited guests can access the sandbox.

```
CreateSandboxAndUninvitedConnect:
SetupClients(1);
SetupClients(2);
SetupProviders(1);
SetupOSs(1);
SetupTools(1);
SetupModels(1);
let clientId in set {1}
in
(SelectOS(clientId, 1);
 SelectToolFromRepository(clientId, 1);
 SelectModelFromRepository(clientId, 1);
 LaunchNewSandbox(clientId);
 AccessSandbox(2, 1));
```

Listing 1.12. Uninvited client attempting connection to a sandbox

Several other traces have been analysed, covering the launching of the sandbox, where these traces have confirmed that the permission table does allow access as specified. One of these flows is shown in Listing 1.13, representing a simple scenario of a user with a provider profile launching a new sandbox.

```
CreateSandboxNoModelNoTool:
SetupClients(1);
SetupProviders(1);
SetupOSs(1);
let clientId in set {1}
in
(SelectOS(clientId, 1);
 SelectToolFromRepository(clientId, 1);
 SelectModelFromRepository(clientId, 1);
 LaunchNewSandbox(clientId)
);
```

Listing 1.13. Legitimate user creating a sandbox

4.2   Results and suggestions

The security analysis of the sandbox platform has uncovered potential improvements to the table of roles and profiles. The first suggestion is to explicitly state what roles and

profiles are needed in order for the client to be able to create a new sandbox. Second suggestion is to carry out a consistency check on the roles and profiles after each update of the roles and profiles table, a task well suited for VDM as during the modelling work one such update has been provided by the implementation team with minimal amount of effort to update the model. This first update has been a result of inconsistencies uncovered during encoding of the table into the VDM model.

In total 11 traces have been created, most of them considering only a single value for an input parameter, while a single trace has considered a selection of three values for the input parameter. This lead to creation of 13 tests covering the actions of interest from the roles and profiles table as well as considering an important action of creating a new sandbox. The evaluation of traces has taken less than two seconds, providing a well performing solution for analysis of client permissions. Since the roles and profiles table is continuously updated it is expected that the VDM model pre- and post-conditions on different operation calls will be updated as well in order to carry out an analysis ensuring the security of the sandbox platform.

5   Related Work

Given its popularity, there are several works covering formal verification of cloud systems [5]. In fact, it is not unusual for major cloud vendors to adopt, use, and have internal departments doing research in formal methods and verification tools. Our work is different from the usual in the sense that cloud operators typically do not provide access to the same sandboxed OSs to multiple users at the same time. The work in [16] is an example of the application of formal methods to secure the assets of organisation that wish to use federated cloud systems (systems where several providers and local clouds are combined), but are afraid of compromising the security of their solution. The paper shows how the security of information flow can be analysed. Our work focus is also at the dataflow level, although information security is not our current focus. Our goal is to achieve a solution similar to the one presented in [14] and [3], but generalised beyond network checks and the cloud machinery provisioning respectively. We foresee the development of a security middleware applying checks at the pain points elicited during the platform modelling and auditing tasks. At project proposal phase we envisaged the possibility to specialise and apply malware detection techniques as the one in [13], but the sandbox prototype based on sandboxed OS images is much more suited to generalist and COTS malware checkers.

Sandboxing is a widely used to establish security. Whether we think of the popular browser tabs, an operating system kernel, or a C program memory sandboxes, their security involves a sandbox. Formal methods have been widely used to verify the good application of sandboxing. In [4], the authors propose to extract a browser sandboxing kernel with leak-proof security. Our work does not leverage on theorem proving, and it does not intend to substitute the current implementation with a correct-by-construction one, but that is still a possibility. Our current focus is the usage of lightweight specification in the form of pre/post-conditions and invariants to reason about the system security. In [1] the authors extract an executable COQ model of an hypervisor, a "real-world" C++ implementation of a virtualisation architecture. The executable model is

then used to as an oracle for the expected behaviour. Our work threads along the same lines, we expect to model our sandboxing platform in VDM, from which we expect to derive an oracle and expected traces exhibiting secure and insecure dataflows. In [2], a similar approach is used to program data memory. In all the approaches implementations are expected to be secure, after modelling and using tools in the security audit.

6   Conclusion

This paper reported on findings of use of VDM-SL in security analysis of online (potentially cloud based) collaboration platforms. The system architecture presented in this paper is based on a real life sandboxing platform, a part of a project aimed at collaborative use of model based engineering while preserving the information security and intellectual property. We created a model of the system, considering features provided within a table of roles and profiles by the implementation team. To model this feature set we have considered the mentioned actions and modelled the roles and profiles as pre conditions. We have then analysed several specific scenarios within the model by use of the combinatorial testing. The analysis has uncovered a potential issue, presenting an inconsistency within the intended sandbox creation functionality, specifically an issue within a hierarchy of operation calls, where an action can be requested without specifically defined roles, however it is dependent on another action requiring specific role. These results have been shared with the implementation team, that has used it to more clearly define the table of roles and profiles within the system. The analysis has in total considered 13 tests and could be carried out in seconds.

As a future work we intend to align with the progress of the implementation team and update the model to analyse every update to the table of roles and profiles. Furthermore we plan to extend our model to be able to analyse aspects of federated cloud, where some sandbox servers could exist within an on premises data-center while other servers would exist within a public cloud service. To this end we intend on proposing an access model, i.e. and updated list of roles and profiles, that has been modelled using VDM and share the findings with the implementation team.

References

1. Becker, H., Crespo, J.M., Galowicz, J., Hensel, U., Hirai, Y., Kunz, C., Nakata, K., Sacchini, J.L., Tews, H., Tuerk, T.: Combining mechanized proofs and model-based testing in the formal analysis of a hypervisor. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods. pp. 69–84. Springer International Publishing, Cham (2016)
2. Besson, F., Blazy, S., Dang, A., Jensen, T., Wilke, P.: Compiling sandboxes: Formally verified software fault isolation. In: European Symposium on Programming. pp. 499–524. Springer, Cham (2019)

3. Bleikertz, S., Vogel, C., Groß, T., Mödersheim, S.: Proactive security analysis of changes in virtualized infrastructures. In: Proceedings of the 31st Annual Computer Security Applications Conference. p. 51–60. ACSAC 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2818000.2818034, https://doi.org/10.1145/2818000.2818034

4. Jang, D., Tatlock, Z., Lerner, S.: Establishing browser security guarantees through formal shim verification. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). pp. 113–128. USENIX, Bellevue, WA (2012), https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang

5. Kulik, T., Dongol, B., Larsen, P.G., Macedo, H.D., Schneider, S., Tran-Jørgensen, P.W.V., Woodcock, J.: Formal methods in security survey (in preparation) (2020)

6. Kurita, T., Nakatsugawa, Y.: The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. Intl. Journal of Software and Informatics **3**(2-3), 343–355 (October 2009)

7. Larsen, P.G., Hansen, B.S., Brunn, H., Plat, N., Toetenel, H., Andrews, D.J., Dawes, J., Parkin, G., et al.: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)

8. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science **7**(8), 692–709 (2001)

9. 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). https://doi.org/10.1145/1668862.1668864, http://doi.acm.org/10.1145/1668862.1668864

10. Larsen, P.G., Fitzgerald, J., Brookes, T.: Applying Formal Specification in Industry. IEEE Software **13**(3), 48–56 (May 1996)

11. Larsen, P.G., Macedo, H.D., Fitzgerald, J., Pfeifer, H., Benedikt, M., Tonetta, S., Marguglio, A., Gusmeroli, S., Jr., G.S.: An Online MBSE Collaboration Platform. pp. 263–270. INSTICC, Proceedings of the 10th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - Volume 1: SIMULTECH (July 2020). https://doi.org/10.5220/0009892802630270

12. Larsen, P.G., Soulioti, G., Macedo, H.D., Alifragkis, V., Fitzgerald, J., Livanos, N., Pfeifer, H., Pasquinelli, M., Benedict, M., Thule, C., Tonetta, S., Stritzelberger, B., Marguglio, A., Sutton, L.F., Obstbaum, M., Gusmeroli, S., Beutenmüller, F., Jr., G.S., Wijnands, Q., Talasila, P.: Enabling combining models and tools in an online mbse collaboration platform. In: Model Based Space Systems and Software Engineering (MBSE2020). ESA-ESTEC, Noordwijk, The Netherlands (September 2020)

13. Macedo, H.D., Touili, T.: Mining malware specifications through static reachability analysis. In: European Symposium on Research in Computer Security. pp. 517–535. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

14. Madi, T., Jarraya, Y., Alimohammadifar, A., Majumdar, S., Wang, Y., Pourzandi, M., Wang, L., Debbabi, M.: Isotop: Auditing virtual networks isolation across cloud layers in openstack. ACM Trans. Priv. Secur. **22**(1) (Oct 2018). https://doi.org/10.1145/3267339, https://doi.org/10.1145/3267339

15. Plat, N., Larsen, P.G.: An Overview of the ISO/VDM-SL Standard. Sigplan Notices **27**(8), 76–82 (August 1992)

16. Zeng, W., Koutny, M., Watson, P., Germanos, V.: Formal verification of secure information flow in cloud computing. Journal of Information Security and Applications **27**, 103–116 (2016)

Visual Studio Code VDM Support

Jonas Kjær Rask[1], Frederik Palludan Madsen[1], Nick Battle[2], Hugo Daniel Macedo[1],
and Peter Gorm Larsen[1]

[1] DIGIT, Aarhus University, Department of Engineering,
Finlandsgade 22, 8200 Aarhus N, Denmark
{201507306, 201504477}@post.au.dk, {hdm, pgl}@eng.au.dk
[2] Independent, nick.battle@acm.org

Abstract. How is it possible to significantly improve the Integrated Development Environment (IDE) for VDM from the existing Eclipse-based IDE? The proposal made in this paper is to use language-agnostic protocols such as the Language Server Protocol (LSP) and the Debug Adapter Protocol (DAP) connecting a general editor such as Visual Studio Code with core server functionality. This is demonstrated for editor related features, debugging, and Proof Obligation Generation and Combinatorial Testing support. We also believe that the extension of LSP and DAP will be useful for extending other IDEs for similar specification languages, since using such standard protocols will require less effort to upgrade to modern front-ends for their IDEs.

1 Introduction

The Vienna Development Method (VDM) is one of the approaches to follow, when applying formal methods during the development of computer systems. The method prescribes the development of digital/computer models of the system under development in one of the the VDM specification languages and dialects. If the models are described in an executable subset it is then possible to execute and analyse them to reach a high-fidelity system description, which is then subsequently used to produce code in the programming languages used to operate the system implementation.

To support all the steps involved in the development of a VDM model, there are several tools and Integrated Development Environments (IDEs) supporting the variety of VDM specification languages and dialects to different levels [9]. VDMTools were the first available commercial tool developed in the mid-1990s [7]. Then Overture [8] brought free and open-source support to VDM in 2005, and many others are now available [11, 14].

The Overture tool is one of the most complete and popular IDEs for VDM. It consists of multiple plugins that extend the Eclipse IDE, which appeared in the 2000s. Eclipse is the most popular[1] in its class, but it is being challenged by a new contender, the Visual Studio Code (VS Code)[2] editor, which is based on Electron[3] and fully leverages

---

[1] See https://pypl.github.io/IDE.html
[2] See https://code.visualstudio.com/
[3] See https://www.electronjs.org/.

web technologies. Although not an IDE, VS Code modernized the IDE world, with the introduction of the Language Server Protocol (LSP)[4] and the Debug Adapter Protocol (DAP)[5], which enable the development of IDE features in a general manner. With the two protocols the editor becomes indistinguishable of an IDE, and the implementation code becomes reusable and better maintainable. However, the Overture language core does not implement the LSP and DAP protocols, so the effort required to move towards VS Code is significant, but we expect it to pay off in the long run.

In this paper we investigate the efforts needed to support VDM in VS Code with as much as possible of the support using the standardised protocols LSP and DAP. To support specification lanuage features not found in programming languages we propose an extension to LSP, Specification Language Server Protocol (SLSP). As part of the investigation we have extended VDMJ such that it can be used as a language server that supports both these protocols. The server supports syntax-checking, type-checking and go-to functionality using the LSP protocol, debugging by using the DAP protocol, and Proof Obligation Generation (POG) and Combinatorial Testing (CT) using the SLSP protocol. We have also developed a VS Code extension which connects to the language server in order to provide the language features in the IDE. In addition, we have investigated which features are required to provide full language support for specification languages and which of these are supported by the protocols.

Our work departs from a previous proof of concept, and adds further support for the LSP protocol, support for all the VDM dialects, support for the DAP protocol and support for the SLSP protocol. In addition, this is the first research paper on the topic, and we foresee more publications to emerge from the works towards fully supporting VDM development using VS Code. We believe the resulting extensions will become the next in line supporting VDM development. A modern and robust IDE, which development starts now.

The remaining parts of this paper starts with an overview of the background necessary to understand this paper in Section 2. Afterwards Section 3 explains how the standard protocols LSP and DAP have been used to implement the core of the VS Code support for VDM. This is followed by Section 4 which is evaluating the efforts that has been conducted. In Section 5 we describe related work. Finally, Section 6 provides concluding remarks and future work.

2 Background

In this section, we introduce VDMJ which is a tool that provides language features for VDM. We further describe the development tool VS Code, which is used as the Graphical User Interface (GUI) for integration with the language server. Finally, we describe the standardised protocols LSP and DAP used to decouple the language features from the development tool.

---

[4] See https://microsoft.github.io/language-server-protocol/.

[5] See https://microsoft.github.io/debug-adapter-protocol/.

2.1   VDMJ

VDMJ [1] is a command-line tool written in Java, that provides basic language support for the VDM dialects VDM-SL, VDM++ and VDM-RT. It includes a parser, a type checker, an interpreter, a debugger, a proof obligation generator and a combinatorial test generator with coverage recording, as well as VDMUnit support for automatic testing and user definable annotations. These are implemented using an extensible Abstract Syntax Tree (AST) analysed using a visitor framework [4].

Using VDMJ for the language server allows parts of the language support to be reused. However, many features of VDMJ are not supported by standardised protocols, thus only a subset of the functionality is used in the language server.

2.2   Visual Studio Code

VS Code[6] is a free source code editor. It has built-in support for the programming languages JavaScript and TypeScript and further enables support for other languages through a rich ecosystem of extensions. VS Code uses a folder or workspace system for interacting with a project and a document system for handling the source code files in the project. This allows VS Code to be language-agnostic and delegate language specific functionality to an extension. Given this design reasoning, a need for the standardisation of decoupling between editor and extension has been identified. At the time of writing, three different protocols have been developed for this purpose. Namely LSP[7] (described in Section 2.3), DAP[8] (described in Section 2.4) and Language Server Index Format (LSIF)[9].

2.3   Language Server Protocol

Since most IDEs support a variety of programming languages, IDE developers face the problem of keeping up with ongoing programming language evolution [2]. At the same time language providers are interested in providing as many IDE integrations as possible to serve a broad audience. Consequently, integrating every language, m, in every IDE, n, leads to a $m \times n$ complexity.

The LSP protocol defines a standardised protocol to be used to decouple a language-agnostic development tool (client) and a language-specific server that provides language features like syntax-checking, hover information and code completion. This is illustrated in Figure 1. The client is responsible for managing editing actions without any knowledge of the language and the server validates the correctness of the source code and reports issues and language-specific information to the client. To facilitate this the LSP protocol communicates using language neutral data types such as document references and document positions.

---

[6] See https://code.visualstudio.com/

[7] See https://microsoft.github.io/language-server-protocol/

[8] See https://microsoft.github.io/debug-adapter-protocol/

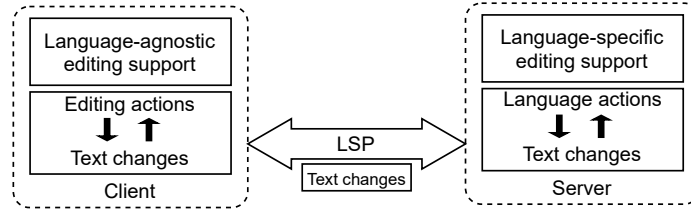[9] See https://microsoft.github.io/language-server-protocol/overviews/lsif/overview/

Fig. 1: LSP approach to language support. Borrowed from [13].

Many tools support the LSP protocol which reduces the time needed to create a client implementation[10]. Furthermore, new development tools only have to support the protocol, which can be done with little effort compared to native integration [3]. Additionally, as the server is separated from the development tool it can be used for multiple tools. This allows tools to easily support multiple languages and features. Thus, by decoupling the language implementation from the editor integration the complexity is reduced to $m + n$.

2.4 Debug Adapter Protocol

The DAP protocol is a standardised protocol for decoupling IDEs, editors and other development tools from the implementation of a language-specific debugger. The DAP protocol uses language neutral data types, which makes the protocol possible to use for any text-based language. The debug features supported by the protocol includes: different types of breakpoints, variable values, multi-process and thread support, navigation through data structures and more.

To be compatible with existing debugger components, the protocol relies on an intermediary debug adapter component. It is used to wrap one or multiple debuggers, to allow communication using the DAP protocol. The adapter is then part of a two-way communication with a generic debugger component, which is integrated in a given development environment as illustrated in Figure 2. Thus, the protocol reduces a $m \times n$ problem of implementing each language debugger for each development tool into a $m + n$ problem.
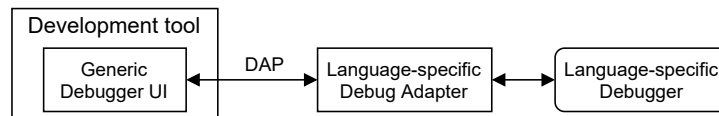

Fig. 2: The decoupled architecture where the DAP protocol is used.

---

[10] See https://microsoft.github.io/language-server-protocol/implementors/tools/

3   Implementing Language Features

The support for VDM in VS Code is accomplished using an extension that communicates with a language server based on the language support found in VDMJ, as illustrated in Figure 3. The standard protocols LSP and DAP are developed with programming languages in mind. This means that the specification language features that are common for these languages are not supported, e. g., POG and CT, as discussed in Section 4.2. To compensate for this we have developed the SLSP protocol, which is introduced in the section Section 3.1. Followed by a description of the components that the VS Code extensions[11] are comprised of and how they have been implemented. Finally, we describe the work put into VDMJ to provide support for the protocols.
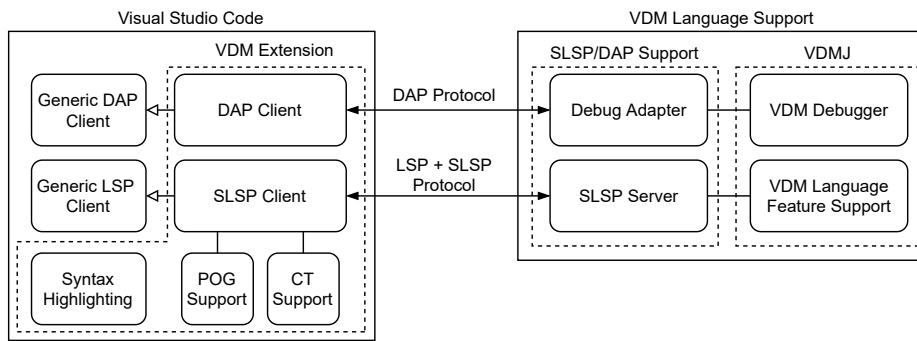

Fig. 3: Overview for the suggested extension architecture

3.1   Specification Language Server Protocol

The purpose of the SLSP protocol is to facilitate support for specification language specific features using the same architecture as for LSP and DAP. The SLSP protocol is an extension to the LSP protocol. This means that it uses the same base protocol as LSP and relies on functionality from LSP such as the synchronisation between client and server. The protocol is developed with the intention that it should also be usable by other languages than VDM and possibly be included in the LSP specification. Hence, the SLSP protocol uses language neutral data types, which allows the client to be language-agnostic enabling it to be used with any server that supports the SLSP protocol. At the time of writing the SLSP protocol defines messages to support POG and CT, and it is used in the VS Code extension for those features.

3.2   Visual Studio Code Extension

VS Code operates with a rich extensibility model that enables users to include support for different programming languages, debuggers and various tools to support the develop-

---

[11] The extensions can be found at: https://github.com/jonaskrask/vdm-vscode

ment workflow, by downloading extensions from the VS Code extensions marketplace[12].
This also enables developers to make extensions that they find missing from the mar-
ketplace, which is how VDM is supported in VS Code; by creating an extension for
each VDM dialect. The extensions include a SLSP client and a DAP client. Besides
communicating with the server the extension also has the responsibility of doing syntax
highlighting as this is not supported in any of the protocols.

As illustrated in Figure 3, VS Code includes generic support for both the LSP and
DAP protocols which allows the client implementation to be carried out with little effort.
The generic protocol support provides the extension with handlers for all the LSP and
DAP messages which include code to synchronise the editor and the server, handling
of the editor navigation and displaying messages from the server. Thus, the extension
provides configuration of the generic protocol support, launching and connecting to the
server, syntax highlighting, and the GUI to support the SLSP features. Figure 4 depicts
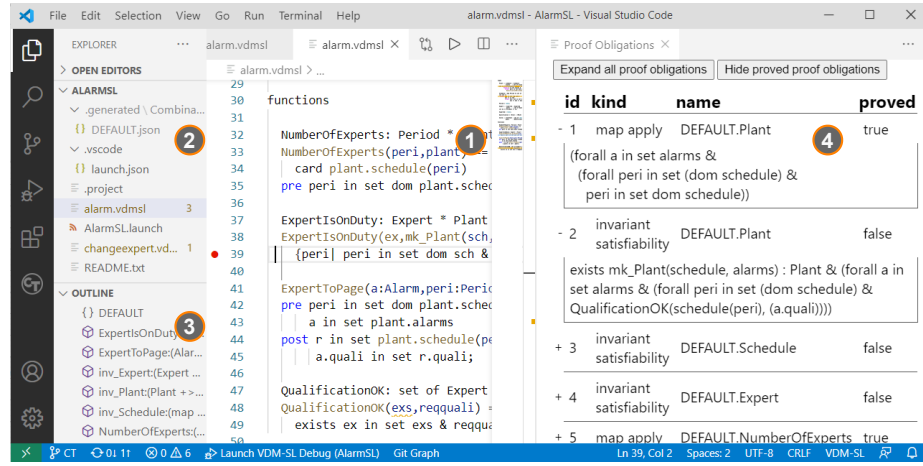the VS Code environment with the VDM-SL extension active.

Fig. 4: VDM-SL VS Code Extension: (1) Main Editor; (2) Project Explorer; (3) File
Outline; (4) Proof Obligation View

Syntax Highlighting: Syntax highlighting in VS Code is handled on the client side as
it is not supported by the protocols. Instead syntax highlighting is performed using Text-
Mate grammars [5]. The TextMate grammars are structured collections of Oniguruma
regular expressions[13], implemented using a JSON schema. The grammar file specifies a
set of rules, that is used with pattern matching to transform the visible text into a list of
tokens and colour these according to a colour scheme.

---

[12] See https://marketplace.visualstudio.com/vscode
[13] See https://github.com/kkos/oniguruma

Specification Language Server Protocol Client: As custom for language extensions the VDM extensions are activated when a file with a matching file format is opened, e. g., opening a '`.vdmsl`' file actives the VDM-SL extension. On activation the client launches the server and connects to it. When connected, the client forwards all relevant information to the server using the LSP protocol. Furthermore, any responses or notifications from the server triggers feedback to the user, which is handled by the generic LSP client integration in VS Code. In addition to setting up the support for LSP the client also provides customised GUI views to support POG and CT. This is not directly supported by VS Code and LSP since these features are not commonly used for traditional programming languages.

The POG view, illustrated in Figure 4, provides a list of the proof obligations for a specification, where expanding an element displays the actual proof obligation. The view is implemented using the VS Code Webview API[14], which is customised using CSS, HTML and JavaScript.

The CT view, illustrated in Figure 5, shows the tests that are generated based on the traces in the specification, these are structured in a tree view. Traces can be fully or partially executed using either an execution filter or executing a test group at a time. A tests execution sequence and result is displayed in a separate tree view. Tests can also be debugged, which is facilitated using the DAP protocol. The CT view is implemented using the VS Code Tree View API[15]. which reduces the complexity of implementing a view.
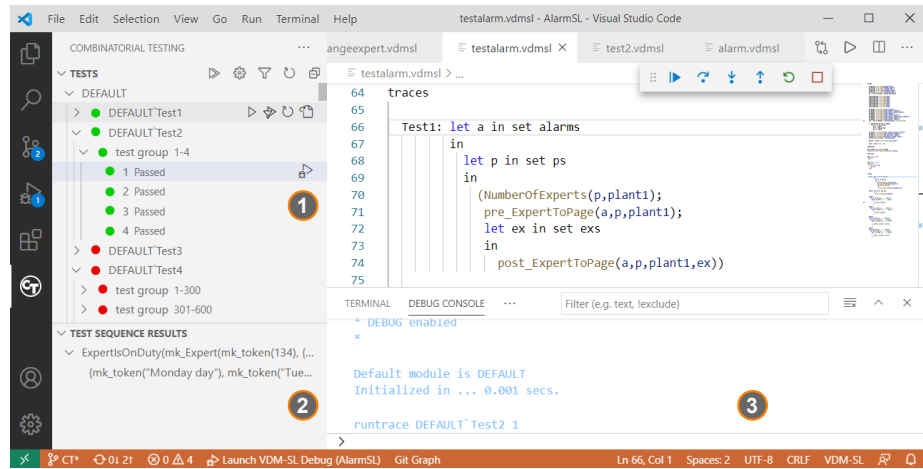

Fig. 5: VDM-SL VS Code Extension: (1) Combinatorial Testing View; (2) Test Result View; (3) Debug Console

---

[14] See https://code.visualstudio.com/api/extension-guides/webview.
[15] See https://code.visualstudio.com/api/extension-guides/tree-view.

Debug Adapter Protocol Client: The DAP protocol support is added to the client by specifying a `DebugAdapterDescriptorFactory` which establishes the connection between the generic debugger client and the server upon starting a debug session. This would normally include launching a DAP server and connect to it. However, as the LSP and DAP server is combined into one we simply connect to the port specified for DAP communication.

3.3   Enabling the use of LSP and DAP in VDMJ

Internally, VDMJ provides a set of language services to enable VDM specifications to be processed, and by default the coordination of these services is handled by a command-line processor. So for example, a Parser service is used to analyse a set of VDM source files to produce an Abstract Syntax Tree (AST); if there are no syntax errors, that AST is further processed by a Type Checker; and if there are no type errors, the tree is further processed by an Interpreter (in a read-eval loop) or by a Proof Obligation Generator. If the Interpreter encounters a runtime problem or a debug breakpoint, it stops and interacts with the command-line to allow the user to examine the stack and variables, for example.

Although the command-line is the default way to interact with VDMJ, this is not assumed in the design. An alternative means to interact is provided using the DBGP protocol[16]. This allows VDMJ to interact with a more sophisticated IDE, such as Overture. DBGP messages are exchanged over a local TCP socket, allowing the IDE to coordinate the execution of expressions within the specification, and to debug using a richer user interface. The VDMJ end of the DBGP socket coordinates the same services as the command-line does, but with requests and responses being exchanged via the socket, using DBGP messages.

So adapting VDMJ to use the LSP and DAP protocols is a matter of creating a new handler to accept socket connections from a client, and process the JSON/RPC messages defined in the respective LSP and DAP standards. Since LSP and DAP are expressed in general language terms, there is generally a natural mapping from abstract concepts in these protocols to the specific case of a VDM specification.

Creating the LSP part of the VDMJ handler was relatively simple. The biggest difference to the existing connection handlers is that LSP is an "editing" protocol. That is, rather than being asked to process a fixed set of unchanging specification files, LSP requires the server to accept the creation and editing of files on the fly. So whereas normally, the Parser would only be called once to process a set of fixed specification files, in the LSP handler it can be called repeatedly as edits are passed from the client. Since the VDMJ parser is relatively efficient, it is responsive enough to re-parse the affected file as edits are made. A decision was taken to only type-check the specification when the client deliberately saves the work (i. e., writes changes to disk). This is the same behaviour as Overture.

Execution and debugging via the DAP protocol required the creation of a new "DebugLink" subclass in VDMJ. The existing command-line and DBGP protocols each extend an abstract DebugLink class, which allows the Interpreter service to interact with an arbitrary client to coordinate a debugging session. In this case, the client is

---

[16] See https://xdebug.org/docs/dbgp

the DAP client. Since the DebugLink was designed to be extended, this was relatively straightforward – in fact the DAPDebugLink is an extension of the command-line class with the protocol aspects changed from textual read/writes to JSON message exchanges. It is about 200 lines of Java. The only complexity in this area is in the case of multi-threaded VDM++ specifications. Special care must be taken to be sure that all of the separate threads and their data are coordinated correctly, with appropriate mutex protection of the (single) communication channel to the DAP client.

In addition to parsing, type-checking and executing a specification, the LSP protocol also enables the server to offer various language services that allow the client to build a more intelligent user interface. For example, the outline of a file (its contents, in terms of the VDM definitions within and their location) can be queried; any name symbol in a file can be used to navigate to that symbol's definition (e. g., moving from a function application to its definition, possibly in a different file); and name-completion services are offered, where the first few characters of a name can be typed, and the server will offer possible completions. These services do not exist in the base VDMJ, and were therefore added as part of the LSP development. They are mostly implemented via visitors (using the VDMJ visitor framework) which process the AST.

The Proof Obligation Generation and Combinatorial Testing features of VDMJ are maked available with the SLSP protocol by adding new "slsp" RPC methods. Internally, the server routes these new requests to the VDMJ components that are able to perform the analyses.

Proof Obligations (POs) can be generated relatively quickly (usually in a fraction of a second, even for hundreds of obligations), so the "slsp/POG/generate" method takes a one-shot approach, generating and returning all the POs in the specification by default, optionally allowing the UI to limit the scope to a sub-folder of specification files or a single file. The returned obligations are represented as VDM source, as a stack of contexts with a primitive obligation at the end. This allows the UI to decide how to represent the indentation of the full obligation without requiring it to understand the VDM AST.

Combinatorial Testing is a more complex problem, requiring multiple interactions between the UI and the server as new "slsp/CT" methods. Unlike Proof Obligations, Combinatorial Tests can expand to millions of test cases and their execution can take many hours. This in turn means that the UI must have some indication of the progress of the execution, and it must also have the ability to cleanly terminate the execution in the event that the test run is taking too long. The base LSP protocol allows for the asynchronous cancellation of an action, and this is implemented in the LSP Server by running CT in a separate thread, allowing the main thread to listen for more interactions from the UI. This complication means that a check has to be added to prevent the user from trying to launch more than one test execution at the same time (which the runtime could not easily support). The server also has to guard against changes to the specification between the expansion of the tests and their subsequent execution. Once a test run is complete, individual tests can be sent to the Interpreter for debugging. This is enabled via the standard DAP protocol, with the addition of a new "runtrace" launch command. So the runtime thinks that the user has started a normal interactive debugging session,

but instead of executing something like "print fac(10)", they are executing "runtrace A'Test 1234", which will debug test number 1234 of the expansion of the A'Test trace.

In terms of packaging, the SLSP/DAP combination of server functionality is in its own jar, separate from (and dependent on) the standard VDMJ jar. The SLSP/DAP jar is about 174Kb. VDMJ itself is about 2.4Mb.

4 Results and Discussion

In this section we discuss the implementation effort carried out in order to support the LSP, SLSP and DAP protocols for VDMJ and the efforts needed on the client side to provide VDM support. This is compared to the efforts necessary to migrate the Overture language features to a different IDE without using the protocols as performed in [14]. Finally, we discuss to which extent it is currently possible to decouple specification language features from development tools using standardised protocols and what is required to have fully decoupled language support using protocols.

4.1 Assessing the Implementation Effort

The implementation effort is quantified by counting the Lines of Code (LoC) used for implementing support for a given protocol and the features it enables. Providing support the protocols for VDMJ requires 7855 LoC, whereas support for VDM in the VS Code extension only requires 1880 LoC where most are used for the new SLSP features. The few lines of code for the extension can mainly be attributed to the generic LSP and DAP protocol implementations exposed by the VS Code API[17].

A detailed overview of the lines needed to implement the language support is shown in Table 1. In the table the 'json' and 'rpc' directory contain a basic JSON/RPC system; 'dap' and 'lsp' are the protocol handlers for each; 'lsp/lspx' is the handler for SLSP; 'vdmj' is the DebugLink and visitors to implement the various features (i.e. these link to the VDMJ jar); and 'workspace' contains the code to maintain the collection of files. For the VS Code extension 'LSP client' and 'DAP client' is the setup of each client; 'POG' and 'CT' is the protocol handlers and GUI support for each feature; 'syntax highlighting' contains the TextMate grammars for VDM; and 'SLSP Protocol' is the definition of the SLSP protocol and expansion of the LSP client.

A comparison between LoC for the VS Code extension, migrating Overture to Emacs (as in [14]) and the Overture IDE (version 3.0.1) not counting the core is seen in Table 2. From the comparison we find that the VS Code extension requires more code than the Emacs migration but Overture consists of far more LoC than both. However, if we only look at the LSP and DAP client support that covers all the normal programming language features they only require 210 LoC. The SLSP parts of the extension are made almost generic, hence these parts can be reused for other specification languages to enable support for the same features in VS Code.

---

[17] See https://code.visualstudio.com/api/references/vscode-api

[18] The LoC is counted using the "VS Code Counter" extension found at: https://marketplace. visualstudio.com/items?itemName=uctakeoff.vscode-counter

Table 1: LoC measures for the files and directories used to support VDM in VS Code[18]

SLSP/DAP support for VDMJ					VS Code Extension				
Directory	**Code**	**Comment**	**Blank**	**Total**	**Feature**	**Code**	**Comment**	**Blank**	**Total**
dap	699	450	185	1334	LSP client	156	16	29	201
json	635	184	139	958	DAP client	54	10	15	79
lsp	1653	625	337	2615	POG	460	51	96	607
lsp/lspx	169	44	30	243	CT	732	105	165	1002
rpc	187	138	56	381	Syntax highlight	374	0	0	374
vdmj	2026	451	392	2869	SLSP protocol	104	167	27	298
workspace	2655	641	545	3841	**Sum**	**1880**	**349**	**332**	**2561**
Sum	**7855**	**2489**	**1654**	**11998**					

For VDMJ to support the protocols it requires considerably more LoC compared to the Emacs approach from [14], which weights against the protocol approach for enabling VDM feature support. However, the protocol solution is a standardisation of the decoupling, hence the server supporting the language features (in this case VDMJ) only has to be implemented once to be used with any development tool supporting the protocols. In comparison, the VS Code extension and migration of the Overture language core to another development tool, as done with Emacs, must be performed for each IDE that is to be supported. Furthermore, the Overture migration depends on Emacs packages, which may not exist in other IDEs or have a different interface. Thus, the analysis of finding suitable packages for another IDE will have to be repeated, possibly making the server solution faster to implement in a new IDE.

Table 2: LoC measure for all IDE related files in the VS Code extension, the Emacs extension and Overture

Project	VS Code (+VDMJ)	Emacs	Overture
Lines of Code	1880 (+7855)	349	>63K

To provide a fair comparison between the three implementation efforts, we compare the features that are available in each. The comparison is done by the same set of features as in [14], this is found in Table 3. As illustrated, the Overture IDE and the Emacs migration both support more features than the VS Code extension. However, some of the features provided by the Emacs migration is only available through a command-line interface and not by a GUI. VS Code does support implementation of integrated command-line interfaces like the Emacs migration, however the implementation efforts related to this are unknown. Providing this kind of command-line interface has intentionally been left out of the VS Code extension, as we wanted to replicate the workflow from Overture.

Table 3: Feature comparison between the VS Code extension, Overture and Emacs. X indicates that a feature is supported. (X) indicates that the feature is only available through a command-line interface, meaning that it is not supported by a graphical user interface.

Features	VS Code	Overture	Emacs
Syntax highlighting	X	X	X
Symbol prettyfication			X
Syntax validation	X	X	X
Evaluation	X	X	(X)
Debugging	X	X	(X)
POG	X	X	(X)
LaTeX report generation		X	(X)
Combinatorial testing	X	X	(X)
Code generation		X	
Auto completion (limited)	X	X	X
Template expansion		X	X
Standard library import		X	

4.2 Protocol Coverage of Language Features

To get an overview of the specification language features that are covered by the protocols, we have examined and grouped the features that are sought after to support specification languages such as VDM, B and Z. This is illustrated in Figure 6, where the language features have been divided into four categories:

1. **Editor**: features that support writing a given specification, e.g. type- and syntax-checking.
2. **Translation**: features that support translating a specification to other formats, e.g. executable code and LaTeX.
3. **Validation**: features that support validation of a specification.
4. **Verification**: features that support verification of a specification.

As a result of the initial implementation and by comparing the features supported by the LSP protocol with the features from Overture, it is found that the protocol is able to support all of the editor related features except syntax highlighting. Thus, specification languages that benefits from the editor features found in Figure 6, can use a direct implementation of the LSP protocol to support this feature set.

The DAP protocol can be used for decoupling the debugging feature as it supports common debug functionality such as different types of breakpoints, variable values and more. This is useful for specification languages that allow execution of specifications.

Additionally, the SLSP extension to LSP facilitates support for both POG and CT. As found in Figure 6 there are still many features related to specification languages that are not supported by protocols. Thus, in order to achieve fully decoupled language server support for VDM one or more protocols must be developed that support the remaining features. From our development of SLSP we believe that it is possible to develop a language-agnostic protocol to support all of these features, which should
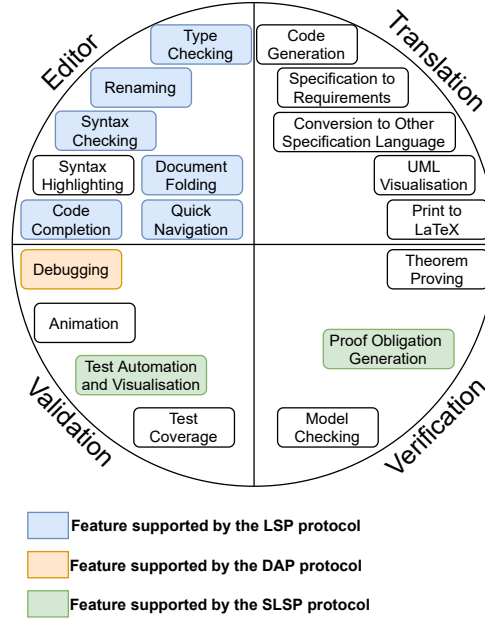
Fig. 6: Specification language features covered by existing protocols

make it possible to apply the protocol to multiple specification languages and potentially increase the industrial uptake of specification languages.

5   Related Work

Support for multiple editors has become common for programming languages, and similar work is now being conducted to achieve the same for modelling and specification languages. Previous work has been focused on providing VDM support for specific platforms, such as Emacs [14], VDMPad [11] and Overture Web IDE [12]. For programming languages it is common to provide language support using standardised protocols to contain the language support in a language-specific server that can be used with multiple language-agnostic clients. This tendency is also becoming apparent for other computer-based languages, such as domain-specific languages (DSL) [3], graphical modelling [13] and formal specification languages such as PVS [10], Dafny [6] and our work for VDM.

Tran and Kulik [14] is able to migrate the Overture core from the Eclipse IDE to Emacs. This is carried out using as many Emacs packages as possible which allows them to provide VDM support with very few additional LoC. This is interesting as it provides knowledge about how few LoC is necessary, which reduces the potential gain from a generalised solution, i. e., if the generalised solution requires a lot of effort you could argue that you might as well have several small dedicated solutions.

Both Oda et al. [11] and Reimer and Saaby [12] presents a web IDE for VDM, based on VDMJ and Overture respectively. Even though both are web based they do not utilise the same language core, thus not providing the same set of language features. Both solutions can probably benefit from unifying the efforts towards shared language support, this could be facilitated using the LSP, DAP and SLSP protocols. Also, VS Code is build using HTML, CSS, and JavaScript which allows the IDE to be launched in a web environment, e. g., Coder[19]. Thus, adding to the number of web IDEs for VDM.

Masci and Muñoz [10] provides support for PVS using the LSP protocol and VS Code. They too use the extensibility of the protocol to provide client-server support for non-LSP features, such as POG and theorem proving. These features are facilitated using a PVS-specific protocol. It would be interesting to investigate if the SLSP protocol could also be used in their case, and what changes may have to be made. This could make way for an addition to the LSP protocol that can be used for a variety of specification languages and possibly be incorporated in the official LSP specification.

6   Concluding Remarks and Future Work

In this paper we have investigated the efforts needed to support VDM in VS Code using the standardised protocols LSP and DAP and our LSP extension SLSP. Also, we evaluated to which extent the VDM language features can be covered by the protocols.

The implementation efforts were estimated using a LoC measure. In total, the VS Code extension consists of 1880 LoC, where only 210 are related to the support of the standardised protocols. This is possible because VS Code provides generic modules for supporting the protocols. Similar generic support is found in other IDEs, we therefore believe that the implementation efforts for these IDEs are comparable to our results. The protocol support for VDMJ totals 7855 LoC which we find appropriate as the server allows the language support to be easily reused in other IDEs as they only have to adhere to the protocols to use the server, which requires little effort compared to creating native support for VDM.

In the language feature coverage evaluation we find that only a subset of the VDM language features are covered by standard protocols. To support features related to specification languages, such as POG and CT we propose the protocol extension SLSP.

Going forward we believe that it would be beneficial to also support other specification language features related to validation, verification and translation, using a similar architecture to decouple the language features from the development tool using a language-agnostic protocol. This would make it easier for multiple development tools to support specification languages, reduce the long term efforts needed to maintain language support, and increase the uptake of specification languages by allowing users to use their preferred development tool.

---

[19] See https://coder.com/.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Bünder, H.: Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In: MODELSWARD. pp. 129–140 (2019)
3. Bünder, H., Kuchen, H.: Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol. In: Hammoudi, S., Pires, L.F., Selić, B. (eds.) Model-Driven Engineering and Software Development. pp. 225–245. Springer International Publishing, Cham (2020)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
5. Gray, J.E.: Textmate: Power Editing for Everyone. Pragmatic Bookshelf (2007)
6. Hess, M., Kistler, T.: Dafny Language Server Redesign. Ph.D. thesis, HSR Hochschule für Technik Rapperswil (2019)
7. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
8. 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
9. Larsen, P.G., Fitzgerald, J.: The evolution of VDM tools from the 1990s to 2015 and the influence of CAMILA. Journal of Logical and Algebraic Methods in Programming 85(5, Part 2), 985 – 998 (2016), http://www.sciencedirect.com/science/article/pii/S2352220815000954, articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday
10. Masci, P., Muñoz, C.A.: An integrated development environment for the prototype verification system. Electronic Proceedings in Theoretical Computer Science 310, 35–49 (Dec 2019), http://dx.doi.org/10.4204/EPTCS.310.5
11. Oda, T., Araki, K., Larsen, P.G.: VDMPad: A Lightweight IDE for Exploratory VDM-SL Specification. In: Proceedings of the Third FME Workshop on Formal Methods in Software Engineering. p. 33–39. Formalise '15, IEEE Press (2015)
12. Reimer, R.S., Saaby, K.D.: An Open-Source Web IDE for VDM-SL. Master's thesis, Department of Engineering, Aarhus University, Denmark (May 2016)
13. Rodriguez-Echeverria, R., Izquierdo, J.L.C., Wimmer, M., Cabot, J.: Towards a Language Server Protocol Infrastructure for Graphical Modeling. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. p. 370–380. MODELS '18, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3239372.3239383
14. Tran-Jørgensen, P., Kulik, T.: Migrating Overture to a different IDE. In: Gamble, C., Diogo Couto, L. (eds.) Proceedings of the 17th Overture Workshop. pp. 32–47. No. CS-TR- 1530 - 2019 in Technical Report Series, Newcastle University (2019), http://overturetool.org/workshops/17th-overture-workshop.html, null ; Conference date: 07-10-2019 Through 11-10-2019

Tuning Robotti: the Machine-assisted Exploration of Parameter Spaces in Multi-Models of a Cyber-Physical System

Sergiy Bogomolov[1], John Fitzgerald[1], Frederik F. Foldager[2], Carl Gamble[1], Peter Gorm Larsen[3], Kenneth Pierce[1], Paulius Stankaitis[1], and Ben Wooding[1]

[1] School of Computing, Newcastle University, United Kingdom
`Sergiy.Bogomolov,John.Fitzgerald,Carl.Gamble@ncl.ac.uk`
`Ken.Pierce,P.Stankaitis,B.Wooding1@ncl.ac.uk`
[2] Agro Intelligence, Agro Food Park 13, 8200 Aarhus N, Denmark,
`ffo@agrointelli.com`
[3] DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark
`pgl@eng.au.dk`

Abstract. We describe a pilot study that aims to evaluate a systematic approach to tuning the design parameters of Cyber-Physical Systems by using machine-assisted Design Space Exploration supported by the INTO-CPS Toolchain. The study focuses on the design of a modified controller for a semi-autonomous agricultural robot. Beginning from a candidate multi-model built from VDM and 20-sim, the study uses data derived from test runs of the robot to tune selected parameters of a Continuous-Time model of the robot's physics in order to improve fidelity. The study raises questions around support for the selection of parameters for tuning, selection of test scenarios, and the ranking of results. We identify further work to complete the design loop by assessing the consequences of introducing a modified controller and deploying it in further field tests.

1 Introduction

Cyber-Physical Systems (CPSs) integrate computational and physical processes with data and network technologies, creating opportunities for new smart products and services and the digitalisation of industrial processes [24]. CPS design is inherently multi-disciplinary: a developer must bring together the products of diverse disciplines, often from diverse suppliers, with the attendant risks of incompatibilities and misunderstandings which may only come to light in prototyping. Model-based methods of *virtual engineering* [3] have been proposed as a means of addressing these challenges. Among the proposed advantages are the ability to use design models as a basis for optimising parameters by performing systematic Design Space Exploration (DSE) with the goal of reducing the number of prototyping cycles required to converge on an optimal design.

A model-based approach to multi-disciplinary CPS design entails the federation of semantically heterogeneous models. We call such federations *multi-models* and the co-ordinated execution of multi-models we term *co-simulation* [16]. Co-simulation allows the exploration of CPS-level behaviours that arise from the interaction between cyber

and physical elements. In practice, this means that open tool chains should allow for the collaborative construction and co-simulation of multi-models. However, there is only limited experience with DSE over such multi-models within CPS product development.

A practical challenge in CPS design is that of estimating the properties of the physical product, and hence ensuring that cyber elements such as control and communication are matched to physical system elements' properties, to deliver the desired CPS-level performance. In this paper, we report progress in a study that aims to pilot the use of DSE to address this challenge. The underlying co-simulation technology is that of the "Integrated Tool Chain for Model-based Design of Cyber-Physical Systems" (INTO-CPS) project [10]. This makes use of the mature Functional Mock-up Interface (FMI) 2.0 standard for co-simulation [6][4]. The core FMI-compliant tool of the INTO-CPS tool chain is a Co-simulation Orchestration Engine supporting both fixed and variable step-size simulations [27].

The ultimate goal of the study is to evaluate the use of DSE over a multi-model to determine optimal control parameters under different operating conditions for an autonomous agricultural field robot, Robotti [13]. The robot is a tool-carrier designed for accommodating a wide range of field operations. By combining the robot with several tools, a large number of load cases emerge. To reduce time to market and the cost of testing diverse configurations in the field, a structured simulation-based approach is needed to address the increasing number of possible configurations in a virtual setting.

The intention is to use DSE to optimise the steering controller, minimising cross-track error by co-simulation of a simple model representing the dynamics of the robot in a continuous-time (CT) formalism and a model representing the steering controller in a discrete-event (DE) formalism [9]. In the work performed to date and reported in this paper, the model of the machine dynamics is tuned experimentally to ensure that it satisfactorily captures the motion of the robot over a series of scenarios (which are intended to exercise the controller rather than reflecting real field-work). Model calibration and fidelity are critical to applicability [15,16].

We briefly discuss related work on DSE in Section 2. The relevant elements of the INTO-CPS tool chain and its underlying co-simulation are summarised in Section 3. The INTO-CPS support for DSE technology is explained in Section 4. The pilot study of the Robotti case is presented in Section 5 and the DSE analysis of it in Section 6. Finally, Section 7 discusses our results so far and future work.

2   Related Work

The techniques of automated DSE originated in the embedded systems domain, focusing on optimum placement of software functionality on constrained hardware. The need for efficient DSE was noted early [7] and applied to integrated circuits (e.g. [2]). The aerospace domain pioneered adoption of modelling and DSE at the larger-scale systems level, e.g. co-design of the Chinook helicopter's integrated circuit and controller [8] and to multi-objective energy-based optimisation [1]. As in agriculture, the component or components being optimised in aerospace are often from a single domain

---

[4] http://fmi-standard.org

(e.g. a turbine [5] or exhaust units [18]), although the objectives being optimised may be from multiple domains. Where DSE is applied with components from different domains, these models still tend to represent only the physical elements of the system (e.g. combined heat and power [25] and hybrid energy systems cells [23]).

A key challenge in DSE is that the size of the design space increases exponentially as the number and range of parameters is increased [17]. Large design spaces can be managed by increasing simulation resources, or by reducing the number of simulations required. Guidelines on experiment design can help engineers identify important factors to focus their searches, for example using the Taguchi Method [14]. This can be supported using techniques such as computational data clustering algorithms [21] and global sensitivity analysis [19].

3 The INTO-CPS Toolchain

INTO-CPS is an open tool chain that facilitates collaborative development and co-simulation for multi-models[5]. Although it supports traceable systems engineering activities from requirements analysis to test, the tool chain's central feature is support for construction and co-simulation of multi-models using FMI.

The tool chain allows requirements to be described using SysML [26] using a profile for CPSs which has been implemented in the Modelio tool [4]. This profile allows the architecture of a CPS to be described, including both computational and physical elements. Based on a profile-conformant model, one can automatically generate FMI descriptions for each constituent model. It is also possible to generate the connections between different Functional Mockup Units (FMUs) for a co-simulation.

The FMI model descriptions can be imported into diverse modelling and simulation tools. Each of these can produce detailed models that, exported as FMUs, can be incorporated into an overall co-simulation as independent simulation units. The element models can either be in the form of DE models or in the form of CT models combined in different ways. Thus, heterogeneous constituent models can then be built around this FMI interface, using the initial model descriptions as a starting point. A Co-simulation Orchestration Engine (COE) allows these constituent models to be evaluated through co-simulation [27]. The COE also allows real software and physical elements to participate in co-simulation alongside models, enabling both Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation.

A web-based INTO-CPS Application has been produced to allow engineers to manage the multi-modelling and co-simulation processes [22]. This can be used to launch the COE, enabling multiple co-simulations to be defined and executed, and the results collated and presented automatically. The tool chain will allow these multiple co-simulations to be defined via DSE.

4 Design Space Exploration in INTO-CPS

DSE is the activity of evaluating multi-models from a collection of alternatives in order to reach a basis for subsequent more detailed design [14]. The set of multi-models under

---

[5] www.into-cps.org

consideration is termed a *design space*. The multi-models within a design space may differ from one another solely in terms of the values of specific *design parameters*, or may be more fundamentally different, for example in terms of structure or component elements. The purpose of DSE is to evaluate these multi-models under a series of co-simulations, in order to arrive at a *ranking* against an *objective* (or *cost function*), i.e., a set of criteria or constraints that are important to the developer, such as cost or performance. A multi-model or group of multi-models selected from a ranking might then form a basis for further design steps. The value of DSE is in prioritising cyber-physical prototypes which may have to be produced and on which expensive tests may have to be performed.

Design spaces for CPSs may be extremely large, and so different DSE techniques aim to explore the space of alternatives rapidly. These include both *open* and *closed*-loop approaches. In this kind of assessment there can be tradeoffs over multiple potentially competing objectives (e.g., speed and energy consumption). To enable the engineer to select optimal configurations it is necessary to have a convincing way to present the results, for example in the form of a *pareto-front*.

Since system-level properties of CPSs are often the result of interactions between computational and physical processes, it is worth noting that tradeoffs can be made across these domains. For example, one might want to examine any tradeoff between different error detection and recovery strategies in supervisory controller code and, say, the performance and cost of sensors. The ability to do this over a multi-model is one of the advantages of taking a multi-disciplinary co-simulation approach.

The INTO-CPS tool chain supports DSE by running co-simulations on selected points in the design space of interest. Both open loop (exhaustive search) and closed loop (generic search or simulated annealing) approaches can be supported in exploring alternative solutions [12].

Given system requirements (potentially including constraints for different aspects of a CPS) and a multi-model, a desired multi-model configuration can be create for the INTO-CPS Application. Using scripts implemented in a combination of Java and Python, the DSE analysis can be performed from the INTO-CPS Application, and the ranking of results can be displayed after individual simulations have been conducted. This workflow was followed in the process illustrated later in Figure 4.

INTO-CPS supports cloud deployment of DSE through the open-source CondorHTC framework, and genetic algorithms for reducing simulations [11]. Existing guidelines on experimental design for DSE over multi-models, for example using Taguchi Methods, can also help engineers using INTO-CPS [14].

5 Robotti Pilot Study: Overview and Experimental Setup

5.1 Goal of the Study

The purpose of this pilot study is to assess the viability of DSE as a tool in improving the fidelity of a multi-model, based on real-world data observed during field trials of a prototype system. The study is based on the multi-model of the Robotti unmanned platform developed by Agro Intelligence (Agrointelli) for field operations in commercial agriculture (see Figure 1).

Fig. 1: The Robotti unmanned platform

The challenge we address is that some physical properties can be difficult to estimate during design. These include those affected by changing conditions, such as surface friction and tyre stiffness. In order to develop a useful (multi-)model, we aim to approximate these values by measuring the overall performance of a prototype design and tuning the physical parameters of a simple model such that the modelled behaviour of the application matches the measurements. A number of field tests are carried out in order to characterise the actual motion of the robot as a function of given input trajectories. We then vary the physical parameters of the multi-model via DSE and identify the inputs that result in the closest correlation between the modelled and measured trajectories in corresponding scenarios. Note that we do not undertake modelling of the terramechanics including soil characteristics [28].

The goal is to be able to make a simple, low degree of freedom model with only a few parameters that, to an acceptable degree, describes the motion of the robot. The fundamentals of the CT model are also described in [20]. This allows us to tune the model without knowing and modelling in detail the entire physical system by simply finding the best possible set of parameters to describe this motion. In the end, if we can achieve a validated multi-model of both the CT model describing the dynamics and the DE model describing the controller, we can use this knowledge to provide an estimate of how to adjust the control parameters for the best possible performance based on a future round of DSE.

5.2 Field Tests and Scenarios

The field tests comprised a total of 27 runs across four scenarios. Each scenario is a pattern of control outputs designed to assess the dynamical response of Robotti to different motion profiles. Within each test run, the controller was set to "open loop" mode, where the control outputs change but there is no active control based on feedback. The control outputs and GPS position were recorded for each run. The position of the Robotti in a selection of test runs is shown in Figure 2. The four scenarios are:

Speed step: The speed of each wheel is increased incrementally in lock step, resulting in the Robotti driving straight as in Figure 2a. This test was repeated eight times.

Speed ramp: The speed of each wheel is increased smoothly. This also results in a straight path as in the speed step tests. This test was repeated three times.

Turn ramp: The speed of the right wheel is increased smoothly, while the left wheel is kept constant. This results in the turning motion seen in Figure 2b. This test was repeated eight times.

Sin: The speed of the left and right wheel is increased and decreased to produce the sinusoidal motion seen in Figure 2c. This test was repeated eight times.

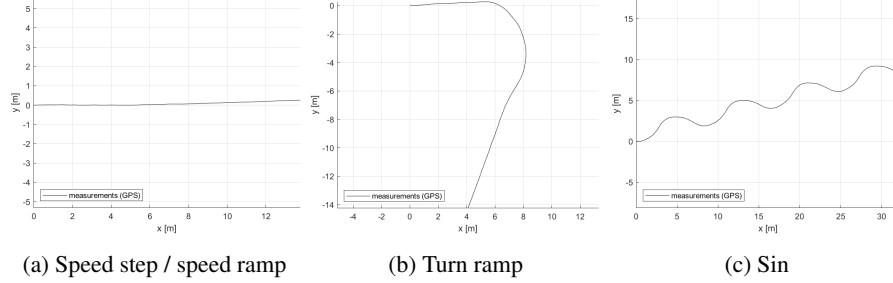(a) Speed step / speed ramp   (b) Turn ramp   (c) Sin

Fig. 2: Path taken by the Robotti during field tests across various scenarios.

5.3 Baseline Robotti Multi-model

The baseline multi-model used for the DSE experiments comprises two FMUs: a CT model of the vehicle realised in 20-sim and a DE model of the open-loop controller realised in VDM-RT and Overture. Figure 3 shows the architecture of the multi-model and the connections between FMUs. The vehicle FMU receives two control signals from the controller FMU, *velocity* and *delta_f*. The former controls the overall velocity of the vehicle and the latter is the steering angle where a value 0 corresponds to driving in a straight line, and positive or negative values indicate right or left turns respectively. Higher absolute values of *delta_f* result in a sharper turn. The vehicle FMU reports the current position and rotation of the Robotti through output signals *x*, *y*, and *theta*.

In turn, the controller FMU sets the control signals as outputs and receives the position and rotation signals from the vehicle as inputs. In this pilot study, these inputs are not used. However this interface means that a closed-loop FMU containing a feedback controller can be easily substituted in future with a closed loop controller. The control outputs for the open-loop field tests were recorded in Comma-Separated Values (CSV) format. In order to perform the same experiments as carried out in the field in a co-simulation, it is necessary to "replay" these control outputs at the appropriate time

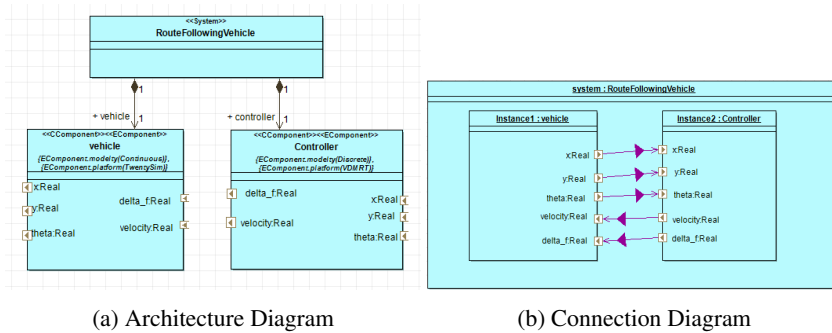(a) Architecture Diagram   (b) Connection Diagram

Fig. 3: SysML diagrams showing the baseline multi-model of the Robotti.

during co-simulation. The CSV library included with the Overture tool is used to load the data, then each data row is compared with the wall clock of the VDM-RT interpreter and the output set appropriately.

6 Robotti Pilot Study: Design Space Exploration

The objective of the DSE was to calibrate Robotti model parameters using results obtained from a physical robot trial. In this section we describe the DSE methodology applied (Section 6.1), the setup (Section 6.2) and results obtained so far (Section 6.3).

6.1 Robotti Pilot Study: DSE Analysis Overview

The INTO-CPS toolchain provides a systematic and flexible method for DSE. Flexibility is achieved by allowing the user to configure DSE analysis with a configuration file (.json format) which defines and orchestrates the exploration process. The methodology also provides an infrastructure for developing and importing external objective functions (as Python scripts) which are used to quantify and rank designs.

Figure 4 shows the workflow and key components of the Robotti DSE. The region in the dashed box represents an existing INTO-CPS DSE infrastructure which contains (and co-simulates) the Robotti multi-model (MM) and a series of manoeuvre scenarios (SN) with corresponding steering inputs (steering_inputs$[s_n]$.csv) and recorded positions from the field trial (gps_position$[s_n]$.csv) for a scenario s_n. Computing a cost of the specific design is performed by objective function script (Algorithm 1) which compares simulated (results$[s_n]$.csv) and recorded positions of the Robotti. The whole DSE process is configured and orchestrated by configuration file (dse_configuration.json) which specifies process parameters such as search algorithm type, multi-model parameters ([cAlpha, mass, mu] which represent total mass of the vehicle, tyre stiffness and surface friction respectively) and their value ranges as well as objective function script file. Once the DSE experiment has been configured, an internal search algorithm simulates the Robotti multi-model with specific parameters and the objective function algorithm computes a cost value of the design. The result of this DSE stage is a dse_results.csv file which maps different simulated multi-model scenarios to a cost value.

The second stage of the Robotti DSE analysis was concerned with finding Robotti multi-model parameters ([cAlpha, mass, mu]) from a generated dse_results.csv data file that would universally fit all simulated scenarios. Finding the best parameter values for all scenarios can be translated to a minimisation problem where an optimisation algorithm attempts to minimise a sum of cost values for different scenarios (with identical [cAlpha, mass, mu] parameters). In this study the optimisation algorithm (OP) has been implemented as an external Python script, which takes dse_results.csv data file and returns best fitting parameter values for all scenarios.

6.2 Robotti Pilot Study: DSE Configuration and Objective Function

Performing DSE entails defining an objective function (Python script) and configuration file which will guide the exploration and ranking processes. The configuration
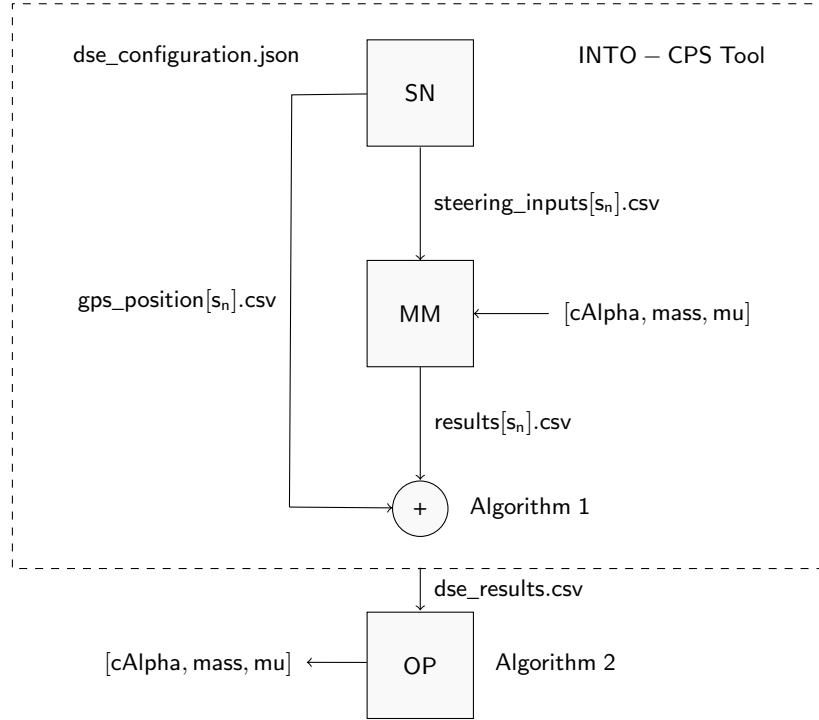
Fig. 4: The DSE analysis workflow overview of the Robotti pilot study

file for our pilot study is shown in Listing 1. It first defines the DSE search algorithm (in our study exhaustive). Next it defines an objective function (external script robottiCrossTrack.py) and simulation outputs used to produce an objective cost value. It then (line 10) specifies the ranges of design parameters which will be used by the exhaustive algorithm to generate different designs. Finally it gives the names of the scenarios to be used.

Listing 1: Robotti pilot study DSE configuration file

```
 1  {
 2  "algorithm": {"type":"exhaustive"},
 3
 4  "objectiveDefinitions": {"externalScripts":{ "robottiCrossTrack":
 5  {"scriptFile":"robottiCrossTrack.py", "scriptParameters":{
 6  "1":"time",
 7  "2":"{Robotti}.RobottiInstance.y3_out",
 8  "3":"{Robotti}.RobottiInstance.y4_out"}
 9  },
10  "parameterConstraints":[],
11  "parameters":{
12  "{Robotti}.RobottiInstance.cAlphaF":[20k,24.5k,29k,33.5k,38k],
13  "{Robotti}.RobottiInstance.mu":[0.3,0.4,0.5,0.6,0.7],
14  "{Robotti}.RobottiInstance.m_robot":[1k,1.5k,2k,2.5k,3k]
15  },
16  "scenarios":["sin1, sin2, sin3, turn_ramp1, turn_ramp2, turn_ramp3,
        speed_ramp1, speed_ramp2, speed_step1, speed_step2, speed_step3"]
17  }
```

Given that the purpose here is calibration, we aim for minimal distance between measured and simulated results. Hence a root mean squared quadratic error is considered an appropriate cost function to evaluate. It was decided to base the tuning of the Robotti model parameters on the mean deviation between recorded and simulated robot positions for a specific manoeuvre. The deviation measure has been implemented with a root-mean-squared-error metric which measures mean a distance between two series of coordinates and can be expressed as Equation 1.

$$\text{rmse} = \frac{\sum\limits_{i=1}^{n} \sqrt{(x_{i2} - x_{i1})^2 + (y_{i2} - y_{i1})^2}}{n} \qquad (1)$$

The measure was implemented as a Python script (described in Algorithm 1) which imports gps_pos.csv and simulation result results.csv files. The cross_track_error function then firstly applies a numerator part of the root-mean-squared-error measure for corresponding rows of imported files and once every position entry of the gps_pos.csv has been compared the function computes and returns a mean error value. The measured objective value with the name of objective name are then stored in the objective.json file with an internal tool function writeObjectiveToOutfile.

Algorithm 1 Root mean squared error objective function algorithm script

---

```
 1: [Skeleton Code]
 2:
 3: import gps_pos.csv as gps
 4: import results.csv as results
 5:
 6: def cross_track_error(gps, results) :
 7:     result = [ ]
 8:     for p in range[0, len(gps)]
 9:         result.append[sqrt((gps[x_p] − results[x_p])² + (gps[y_p] − results[y_p])²)]
10:     mean_error = sum(result)/len(gps)
11:     return mean_error
12:
13: objective_result = cross_track_error(gps, results)
14: writeObjectiveToOutfile(objectives.json, objectiveName, objective_result)
```

Once a cost value has been computed for all permutations of parameters and scenarios, all data is collated and a dse_results.csv file is produced, which in the Robotti DSE study contained the mapping between different simulated multi-model scenarios and a cost value.

6.3 Robotti Pilot Study: Optimum Parameter Search and Results

The main objective of this exercise is finding the best fitting parameter value of the model given the different simulated and measured scenarios. The problem of finding the best fitting parameters can be considered as optimisation (function minimization) problem where a function in Robotti pilot study to be minimised maps parameter values to a mean cross track value error computed by Algorithm 1.

The function we intend to minimise is given in Equation 1 where $\mathcal{E}_s$ is a set of mean track error functions constructed from the simulated scenarios and parameterised by a scenario $s \in \{sin1, sin2, sin3 \cdots\}$ and constrained domain $dom(\mathcal{E})$. The domain of the function $\mathcal{E}_s$ is identical to parameter ranges used by the exhaustive search algorithm (see lines 12-14 in Listing 1)

$$f^- = arg \ min \sum_{s \in S} \mathcal{E}_s(\mathsf{cAlpha}, \mathsf{mass}, \mathsf{mu}) \tag{2}$$

The minimization function has been implemented as an external Python script and is described in Algorithm 2. The main function of the algorithm searchParameters explores all permutations of parameter values and for each permutation calls getResults function which takes specific parameter values and returns a sum of mean squared error values of all scenarios with given parameters. If the returned total_error_value is smaller than the current value of variable minimum then a new minimum value is assigned and optimal parameter values params are updated. The algorithm is exhaustive and will terminate once all parameter value permutations have been explored.

Algorithm 2 Total mean track error value minimisation algorithm

---

```
 1: def getResults(cAlpha, mass, mu) :
 2:       total_error_value = 0.0
 3:         for s in [s₀, s₁, ⋯ sₙ]
 4:             total_error_value + = ℰₛ(cAlpha, mass, mu)
 5:       return total_error_value
 6:
 7: def searchParameters() :
 8:       minimum = init
 9:       cAlphaF = [20000, 24500, 29000, 33500, 38000]
10:       mass = [1000, 1500, 2000, 2500, 3000]
11:       mu = [0.3, 0.4, 0.5, 0.6, 0.7]
12:
13:       for i₀ in range[0, len(cAlphaF)]
14:           for i₁ in range[0, len(mass)]
15:               for i₂ in range[0, len(mu)]
16:                   if(getResults(cAlphaF[i₀], mass[i₁], mu[i₂]) < minimum) :
17:                       minimum = getResults(cAlphaF[i₀], mass[i₁], mu[i₂])
18:                       parameters = cAlphaF[i₀], mass[i₁], mu[i₂]
19:
20:       return (minimum, parameters)
```

The results of the Robotti DSE exercise are summarised in Table 1 and Figures 5 and 6. In total 12 different Robotti manoeuvering scenarios have been considered where each scenario was simulated with 125 parameter [cAlpha, mass, mu] value variations giving us 1500 data points in dse_results.csv file. The optimisation algorithm found the best fitting parameter values Robotti multi-model are CAlphaF = 38000, mass = 1000, μ = 0.3 resulting in a total mean cross track error of 17.865.

In Table 1 for each scenario we provide the smallest and largest possible error values as well as average error over 125 parameter value permutations. The computed parameters for 4 scenarios out of 12 give a below average mean squared error value and 5

slightly above. The interesting case was a speed_step scenario (accelerating and braking in a straight line) where, as expected, a mean squared error value was unaffected by parameter variations and therefore had no effect on tuning Robotti parameters. From Figure 5 we can also notice how different parameter values were affecting four different groups of scenarios and we can notice that sin1 and turn_ramp1 manoeuvres correlate similarly with changing parameter values, whereas speed_ramp has the lowest mean squared error when mu is 0.3 and only very slightly was affected by varying other parameters. Figure 6 illustrates another correlation, which includes the scenario groups sin, speed_ramp, turn_ramp and sub-scenarios (e.g. sin1, sin2) in the individual sub-plots. With this figure we illustrate that the search space for the globally best parameters, where optimised parameters would be universally best was very small and for most scenario groups parameters affect mean squared error unevenly.

scenario	min_error	max_error	average_error	computed
sin1	1.188	1.371	1.323	1.318 (-5E-3)
sin2	1.680	1.960	1.884	1.891 (+7e-3)
sin3	2.139	9.218	4.561	5.486 (+9.25e-1)
speed_step1	0.742	0.742	0.742	0.742 (0)
speed_step2	0.578	0.578	0.578	0.578 (0)
speed_step3	0.482	0.482	0.482	0.482 (0)
turn_ramp	2.056	2.222	2.188	2.194 (+6e-3)
turn_ramp2	1.408	1.560	1.525	1.542 (+1.7e-2)
turn_ramp3	2.233	2.497	2.431	2.456 (+2.5e-2)
speed_ramp1	0.307	5.120	3.244	0.312 (-2.932)
speed_ramp2	0.367	3.965	3.328	0.430 (-2.898)
speed_ramp3	0.431	1.460	1.214	0.431 (-1.1709)

Table 1: The summary of optimisation results and best fitting parameters: mean_cross_track_error$_{total}$ = 17.865 with CAlphaF = 38000, mass = 1000, μ = 0.3

7 Discussion

We have so far undertaken the data engineering needed to perform co-simulation on the Robotti CT model, and have demonstrated DSE-based tuning of this model over three design parameters against an objective function based on mean deviation between the model and the field data. The study, though so far incomplete, has proved technically viable for this scale of multi-model and volume of data. We here consider two main discussion areas based on the experience gained with DSE, INTO-CPS and the Robotti.

Firstly, as with all tools, there is a need for constant updating to stay relevant and useful in a changing technological landscape. While using INTO-CPS, we found difficulties in two main areas. First, INTO-CPS requires Python 2.7 which for most computers required backdating from the more recent Python 3, which is not backwards compatible. A real work-around was necessary to run INTO-CPS tools. Second, in ranking the data, the Pareto Method is built into the toolchain, it ranks simulations against two parameters to provide a Pareto front of optimal values. For any simulations that want to rank against a single parameter the functionality was not simple to find. We also

(a) sin1 Scenario

(b) turn_ramp1 Scenario

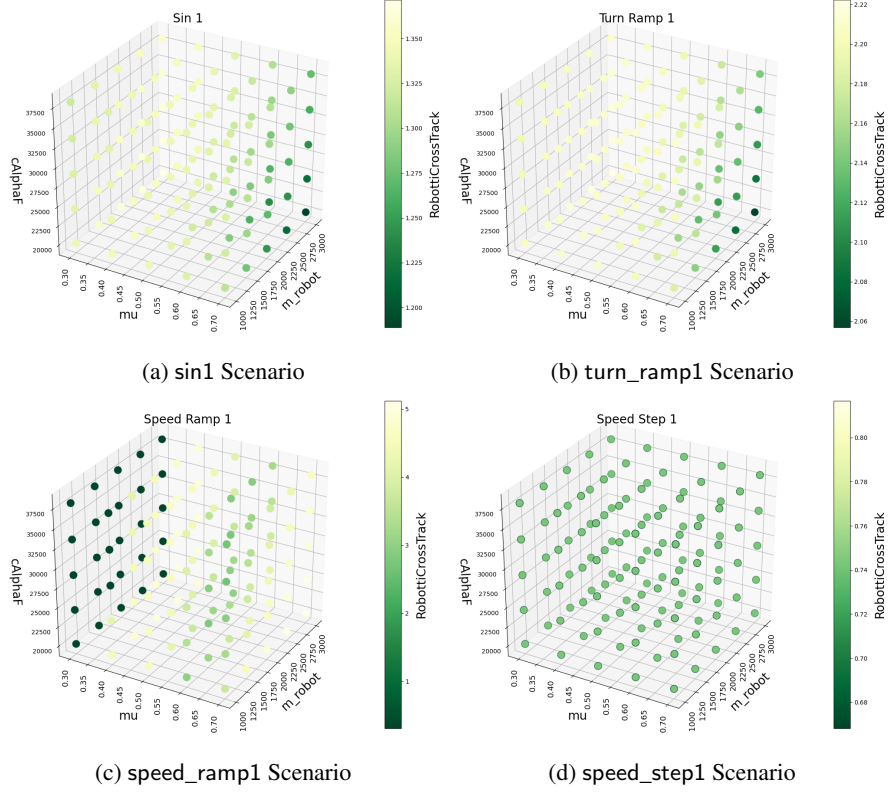(c) speed_ramp1 Scenario

(d) speed_step1 Scenario

Fig. 5: Mean cross track error variation of scenarios against different parameter values

wanted to compare different scenarios in the same set of ranking which is also not a feature readily available in INTO-CPS. In the end we resorted to using Pareto with a second parameter, max cross track error, but as we were not interested in this parameter we ended up writing our own scripts for ranking and finding the optimal parameters across the scenarios.

Secondly, although parameter selection is important when considering the insights that DSE can offer, seeking to optimise performance through DSE, it is equally important to consider the scenarios. For example, in the sin scenarios, all three parameters we looked at affect the cross track error. However, none of them had an impact in the speed_step scenarios and whichever parameter combination was selected the same cross track error was measured. It was theorised that the parameters we discussed were only relevant when the robot would be turning, hence the largest impact appears for the sin wave which has many turns and no impact was felt on the speed_step which has no turning. To improve our results, it is necessary to consider the parameters which constantly affect the robot, not just when turning, so that the most accurate behaviour can be predicted.

In terms of further work, the next step in the pilot study is to complete the co-simulation of the ADRC controller with the plant model that has been tuned so far,
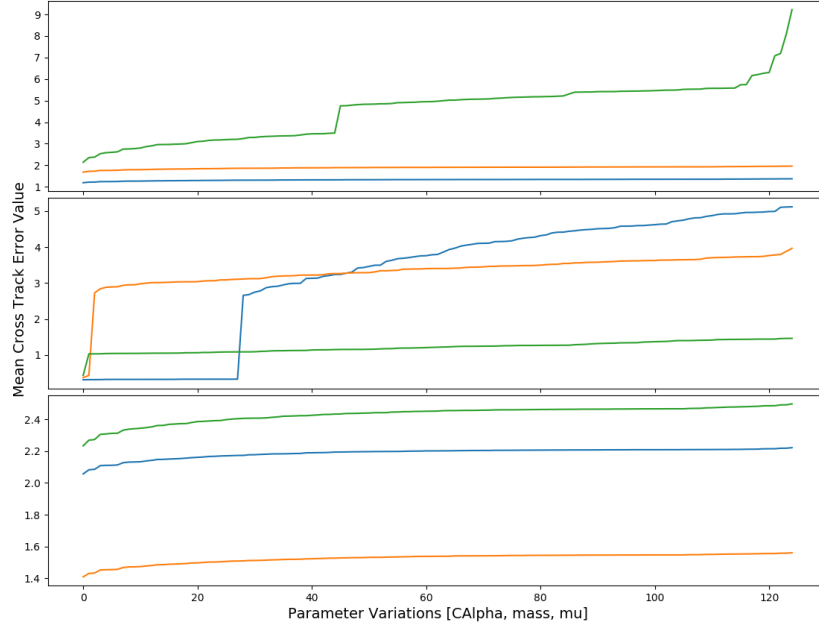
Fig. 6: Mean cross track error for different scenario groups (from top sinN, speed_rampN, turn_rampN) and scenarios (blue (N=1), orange (N=2), green (N=3))

and confirm the extent to which this is faithful to data gathered from field trials. An ADRC controller in itself has about 6 parameters, so exhaustive coverage may not be available within available resources, suggesting the use of genetic algorithms to select co-simulation runs. Aside from the tooling issues above, a priority for future work might be strengthening support for managing scenarios, and more user-friendly methods for generating custom rankings.

14

References

1. Ahmadi, P., Rosen, M.A., Dincer, I.: Multi-objective exergy-based optimization of a polygeneration energy system using an evolutionary algorithm. Energy 46(1), 21 – 31 (2012), http://www.sciencedirect.com/science/article/pii/S0360544212000990, energy and Exergy Modelling of Advance Energy Systems

2. Apvrille, L., Muhammad, W., Ameur-Boulifa, R., Coudert, S., Pacalet, R.: A uml-based environment for system design space exploration. In: 2006 13th IEEE International Conference on Electronics, Circuits and Systems. pp. 1272–1275 (Dec 2006)

3. Van der Auweraer, H., Anthonis, J., De Bruyne, S., Leuridan, J.: Virtual engineering at work: the challenges for designing mechatronic products. Engineering with Computers 29(3), 389–408 (Jul 2013)

4. Bagnato, A., Brosse, E., Quadri, I., Sadovykh, A.: The INTO-CPS Cyber-Physical System Profile. In: DeCPS Workshop on Challenges and New Approaches for Dependable and Cyber-Physical System Engineering Focus on Transportation of the Future. Vienna, Austria (June 2017)

5. Barzegar Avval, H., Ahmadi, P., Ghaffarizadeh, A., Saidi, M.H.: Thermo-economic-environmental multi-objective optimization of a gas turbine power plant with preheater using evolutionary algorithm. International Journal of Energy Research 35(5), 389–403 (2011), https://onlinelibrary.wiley.com/doi/abs/10.1002/er.1696

6. Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J., Wolf, S., Gmbh, G.I.T.I., Oberpfaffenhofen, D.L.R.: The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In: 8th International Modelica Conference. pp. 105–114. Munich, Germany (September 2011)

7. Chou, P., Ortega, R.B., Borriello, G.: Interface co-synthesis techniques for embedded systems. In: Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design. pp. 280–287. ICCAD '95, IEEE Computer Society, Washington, DC, USA (1995), http://dl.acm.org/citation.cfm?id=224841.225052

8. Chou, P.H., Ortega, R.B., Borriello, G.: The chinook hardware/software co-synthesis system. In: Proceedings of the 8th International Symposium on System Synthesis. pp. 22–27. ISSS '95, ACM, New York, NY, USA (1995), http://doi.acm.org/10.1145/224486.224491

9. Christiansen, M.P., Larsen, P.G., Jørgensen, R.N.: Robotic Design Choice Overview using Co-simulation and Design Space Exploration. Robotics pp. 398–421 (October 2015)

10. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)

11. Fitzgerald, J., Gamble, C., Payne, R., Lam, B.: Exploring the Cyber-Physical Design Space. In: Proc. INCOSE Intl. Symp. on Systems Engineering. vol. 27, pp. 371–385. Adelaide, Australia (2017), http://dx.doi.org/10.1002/j.2334-5837.2017.00366.x

12. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative model-based systems engineering for cyber-physical systems, with a building automation case study. INCOSE International Symposium 26(1), 817–832 (2016)

13. Foldager, F., Balling, O., Gamble, C., Larsen, P.G., Boel, M., Green, O.: Design Space Exploration in the Development of Agricultural Robots. In: AgEng conference. pp. 314–321. Wageningen, The Netherlands (July 2018)

14. Gamble, C., Pierce, K.: Design space exploration for embedded systems using co-simulation. In: Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.) Collaborative Design for Embedded Systems, pp. 199–222. Springer Berlin Heidelberg (2014)
15. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: State of the art. Tech. rep. (feb 2017), `http://arxiv.org/abs/1702.00686`
16. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. ACM Comput. Surv. 51(3), 49:1–49:33 (May 2018)
17. Kang, E., Jackson, E., Schulte, W.: An approach for effective design space exploration. In: Calinescu, R., Jackson, E. (eds.) Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems. pp. 33–54. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
18. Karaali, R., Ärĭlhan Tekin ÃŰztÃijrk: Thermoeconomic optimization of gas turbine cogeneration plants. Energy 80, 474 – 485 (2015), `http://www.sciencedirect.com/science/article/pii/S0360544214013619`
19. Kucherenko, S., Feil, B., Shah, N., Mauntz, W.: The identification of model effective dimensions using global sensitivity analysis. Reliability Engineering & System Safety 96(4), 440 – 449 (2011), `http://www.sciencedirect.com/science/article/pii/S0951832010002437`
20. Legaard, C.M., Gomes, C., Larsen, P.G., Foldager, F.F.: Rapid Prototyping of Self-Adaptive-Systems using Python Functional Mockup Units. In: 2020 Summer Simulation Conference. SummerSim '20, ACM New York, NY, USA, Virtual event (2020)
21. Li, Q., Mahmoud, E., Michael, R.: A combinatorial approach to tradespace exploration of complex systems: A cubesat case study. INCOSE International Symposium 27(1), 763–779 (2017), `https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2017.00392.x`
22. Macedo, H.D., Rasmussen, M.B., Thule, C., Larsen, P.G.: Migrating the INTO-CPS Application to the Cloud. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) Formal Methods. FM 2019 International Workshops. pp. 254–271. Springer-Verlag, LNCS 12233 (2020)
23. Mamaghani, A.H., Najafi, B., Shirazi, A., Rinaldi, F.: Exergetic, economic, and environmental evaluations and multi-objective optimization of a combined molten carbonate fuelÂăcell-gas turbine system. Applied Thermal Engineering 77, 1 – 11 (2015), `http://www.sciencedirect.com/science/article/pii/S135943111401134X`
24. Reimann, M., Rückriegel, C., et al.: Road2CPS: Priorities and Recommendations for Research and Innovation in Cyber-Physical Systems. Steinbeis-Edition (2017)
25. Sanaye, S., Katebi, A.: 4e analysis and multi objective optimization of a micro gas turbine and solid oxide fuel cell hybrid combined heat and power system. Journal of Power Sources 247, 294 – 306 (2014), `http://www.sciencedirect.com/science/article/pii/S0378775313014158`
26. OMG Systems Modeling Language (OMG SysML^{TM}). Tech. Rep. Version 1.5, Object Management Group (May 2017), http://www.omg.org/spec/SysML/1.5/
27. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The INTO-CPS co-simulation framework. Simulation Modelling Practice and Theory 92, 45 – 61 (2019), `http://www.sciencedirect.com/science/article/pii/S1569190X1830193X`
28. Wong, J.Y., Reece, A.: Prediction of rigid wheel performance based on the analysis of soil-wheel stresses part i. performance of driven rigid wheels. Journal of Terramechanics 4(1), 81 – 98 (1967), `http://www.sciencedirect.com/science/article/pii/002248986790105X`

A Co-Simulation Based Approach for Developing Safety-Critical Systems

Daniella Tola[1] and Peter Gorm Larsen[1]

DIGIT, Department of Engineering, Aarhus University, Aarhus, Denmark
{dt,pgl}@eng.au.dk

Abstract. When developing safety-critical systems it is important to provide evidence that such systems do not pose a significant hazard to their surroundings. One way of providing such evidence is known as safety cases and here the Goal Structuring Notation is one of the most popular notations. Evidence can take many different forms and in this paper we explore whether co-simulation of critical scenarios can be used with advantage as a beneficial approach. This is illustrated with a combine harvester using different tools in a Functional Mockup Interface setting.

Keywords: Goal Structuring Notation · Co-simulation · Functional Mockup Interface.

1   Introduction

With the growing development in technology, the complexity of systems nowadays is increasing. More systems are operating in dynamic environments, where living creatures reside. If these systems fail, they may damage their surroundings, therefore they are called safety-critical systems [16].

It is crucial that such safety-critical systems operate in a safe manner. The increase of software in these types of systems makes the process of ensuring the system safety a challenge. This is due to the complications followed with the integration of software and systems engineering. A typical complication could be incomplete software specifications, where important details or requirements may be missing or stated incorrectly [16].

Developing safety-critical systems is a complex process, where a large amount of time is spent on demonstrating the safety of the system, either by performing physical tests or formal verification [7]. The effort required to develop safety-critical systems using traditional processes makes it hard to keep up with the shrinking time-to-market that is expected of such systems [16]. Performing complete physical tests of a system to demonstrate its safety can be challenging due to the uncontrollable operating environments, such as changes in the surrounding atmosphere, and unethical tests, such as testing a pilot-controlled aircraft during a thunderstorm. Instead other Validation and Verification (V&V) techniques must be used, such as simulation and co-simulation [24, 22]. The focus of this paper is how co-simulation can be incorporated in the development process of safety-critical systems.

The rest of this paper starts with background information for the reader in Section 2. Afterwards, Section 3 gives a brief overview of the developed process, followed by Sections 4 and 5 which go further into detail of the main steps of this process. Finally, Section 6 presents the concluding remarks and the potential future work.

2   Background

This section describes the required background knowledge to understand the subsequent sections in this paper.

2.1   Case Study

The case study is based on a system where the goal is to develop a combine harvester, which can autonomously drive within the boundaries of a crop field and harvest crops, without the need for a driver controlling the vehicle.

Figure 1 illustrates how the environment of the autonomous combine harvester may look, including a number of the sensors that will be used in the system. The figure also demonstrates how animals may hide in between the crops, which in some cases can be difficult to detect. The system can potentially injure or kill living creatures in the field, damage the harvested crops (if the vehicle runs over an animal, the harvested crops will be contaminated), or catch fire [26]. These are few of the reasons that this system is categorised as safety-critical. The motivation for using this system as the case study when researching how to incorporate co-simulation in the development of a safety-critical system, is:

- It is challenging to test the system in the different atmospheric environments where the conditions cannot be controlled, such as extremely windy, rainy, or foggy.
- It is unethical to test the system with living creatures on the field, to determine if the system operates as it should under these specific circumstances.
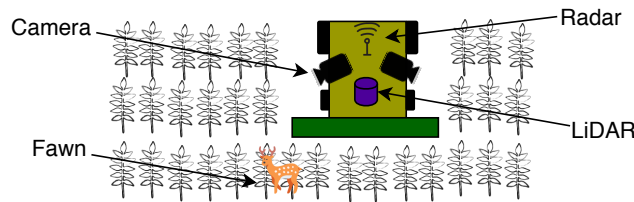

Fig. 1: An illustration of the autonomous combine harvester that is used as the case study in this paper.

2.2 Safety Case

The safety of a safety-critical system must typically be demonstrated to a government regulatory body, by creating a safety case [23], which is defined as [3]:

> "*A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.*"

The three main elements used in a safety case are [3, 15]:

Claim: a safety requirement or claim about the system, for example "System X is safe".
Evidence: used to support the claims about the safety of the system, for example physical tests of the system operating safely in specific environmental conditions.
Argument: used to describe how the evidence supports the safety claims.

The main stages of a safety case, that are relevant for this paper, are illustrated in Figure 2. The information obtained at each stage is used in the following stages. For example, the system description can be used in the process of deriving hazards. A hazard is defined as a system state, where the system may potentially damage its surroundings. It is important to note that a system can be in a hazardous state without causing an accident [23].

The hazard analysis stage consists of identifying hazards and their root causes. Different methods exist for finding root causes of hazards, and usually multiple methods will be used. In this paper we chose to only use one method, *fault-tree analysis*, due to its simplcity and graphical view. It is a top-down approach, starting with a hazard and moving down towards potential events that may lead to the occurrence of the hazard. In the fault-tree analysis approach, events that may lead to a hazard are presented graphically using logical gates, such as AND and OR gates [23].

The risk of each of the identified hazards is then assessed by determining the severity and likelihood of the hazards [6]. The determined risk of each hazard is then used to order the hazards from highest to lowest risk, and thereby determine the acceptable cost of reducing the risk of a hazard [23].

Main Stages of a Safety Case

System Description → Hazard Analysis → Risk Assessment → Safety Objectives → Arguments → Evidence

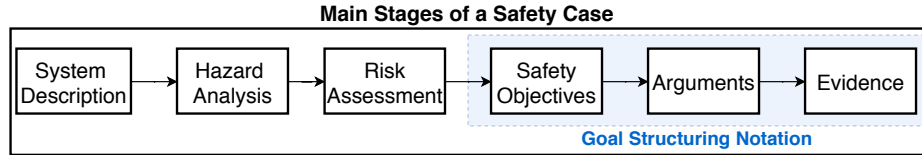**Goal Structuring Notation**

Fig. 2: The relevant main stages of a safety case.

One of the existing safety case processes, is to develop the system first and then create the safety case. The disadvantage of this process is that if safety

issues are found after the development of the system, then a system redesign may be necessary. Therefore, another process has been adopted, which is the phased safety case process, where the system and safety case are both developed simultaneously. This process is beneficial to use when a novel safety argument is being used, which is the case in this paper. There are typically three main phases to be developed in the phased safety case, Preliminary, Interim and Operational. The details of the safety case increase with each phase [14]. We only consider the Preliminary safety case in this paper.

2.3   Goal Structuring Notation

Creating explicit and clear safety arguments using plain text is not always a simple task, and can therefore end in misinterpretations by the regulator. The Goal Structuring Notation (GSN) attempts to solve these potential problems by using a graphical structure, that shows the connections of arguments, safety claims and evidence.

   The relevant GSN elements used in this paper are illustrated in Figure 3, where the *goal*, *solution* and *strategy* are equivalent to the *safety objective*, *evidence* and *argument* in a safety case, respectively [15]. The *away goal* uses a claim from another module to support the argument in the current module [11]. The usage of an *away goal* in a GSN for arguing that a simulation tool adequately models the environmental conditions affecting part of a system, can be found in [24]. This method of argumentation, using an *away goal*, is also used in this paper.
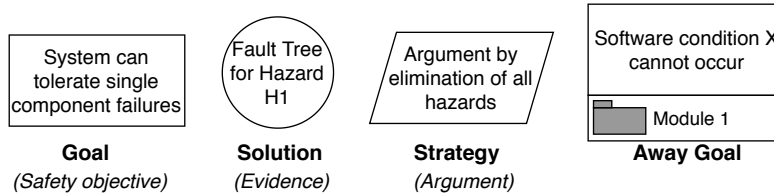

Fig. 3: The relevant elements used in *GSN*. The figure is inspired from [15].

2.4   Modelling and Co-simulation

A model is defined by J. Van Amerongen as [1]:

   "A simplified description of a system, just complex enough to describe or study the phenomena that are relevant for our problem context."

As Amerongan explains, a model should be a simple representation of a system, that is easy enough to understand, but also complex enough to include the important characteristics of the system [1].

Performing simulations of the model with various parameters allows testing the model under different conditions. A challenge of obtaining a simulation of a complete system is that they are composed of several parts; often modelled using different mathematical formalisms. For example, the combine harvester consists of software, electrical and mechanical parts. Co-simulation allows to use models developed in different tools and execute them simultaneously [10], where a co-simulation engine is responsible for exchanging the data between the different models [9].

For the co-simulation engine to be able to communicate with the models, each model must implement an interface, such as the Functional Mock-up Interface (FMI). A model that implements the FMI is called a Functional Mock-up Unit (FMU) [4]. Figure 4 gives an idea of how FMUs developed using different tools can be run in a co-simulation, and also illustrates the main files within in an FMU. An FMU encapsulates a model as a black box with only the necessary parameters, inputs, outputs and binaries exposed in the interface. This allows Original Equipment Manufacturers (OEMs) to share models of their equipment, and at the same time protect their intellectual property [4].
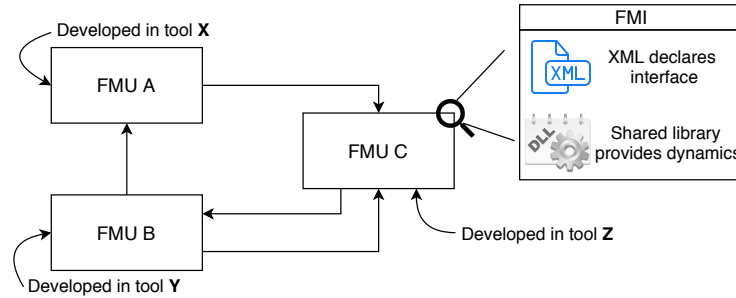

Fig. 4: An illustration of how co-simulation can combine FMUs developed using different tools, and also some of the main files inside an FMU.

3   Process Overview

This section gives a brief overview of the proposed process; more details can be found in [25]. This process provides clear guidelines on how to incorporate co-simulation in the development of safety-critical systems. The process does not yet include how to ensure the validity of the evidence obtained using co-simulations of the system.

The six main stages of the process are illustrated in Figure 5. These stages describe how to start from a hazard analysis and potentially end up with parts of evidence for a safety case. The first two stages are part of the traditional process of creating a safety case, where a safety and hazard analysis is performed,

followed by defining the safety goals of the system. Before proceeding to the third stage, the hazards must be studied to determine which method should be used for creating evidence. How to determine this, is described in the following section. If co-simulation is chosen as an appropriate method for creating evidence, then the next step is to specify test scenarios and create models of the different parts of the system. This is followed by co-simulations of the specified test scenarios, leading to results that if prove the system is safe under the specified conditions, can be used as evidence in the safety case. If the results imply the system is not safe under the specified conditions, then safety improvements must be made and the system must be co-simulated in the failed test scenarios again.
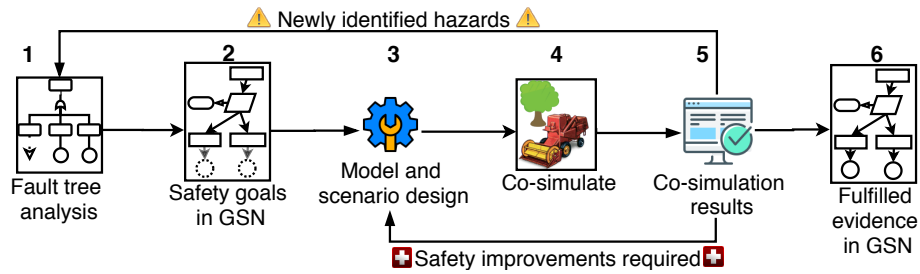

Fig. 5: An overview of the method of developing a safety case using co-simulation.

The contributed stages of the process of creating safety evidence are stages three, four and five, meaning that other safety processes construct evidence using other techniques in these exact stages. It is important to note that determining which elements of the safety evidence that can be constructed using co-simulation results, must be defined in stage two, before moving on to stage three. If it appears that no evidence can be created using co-simulation, then another evidence creation technique must be used, for example mathematical analysis.

The following sections describe how the process was used on the case study, where Section 4 describes the safety analysis, and Section 5 describes how the evidence was created using co-simulation.

4   Safety Case Analysis

This section describes how the safety case analysis of the autonomous combine harvester was performed. The steps illustrated above in Figure 2 are followed, where the first step of describing the system can be found above in Section 2.1. The details of the safety case and system are at a high abstraction level, since a Preliminary safety case is the objective.

Hazards were identified by consulting with an agricultural engineer from AGCO and reading articles. Some of the identified hazards are: *H1: collision with object*, *H3: fire ignition* [26], and *H5: contamination of harvested crops*. A

fault-tree analysis was performed on the hazards to determine potential root causes, see Figure 6a for a simplified fault-tree view. This was followed by a risk analysis to determine which hazards posed the highest risks. The risk analysis showed that hazards *H1* and *H3* posed the highest risks, which are the two hazards considered in this paper.

By analysing these two hazards with regards to how they can be mitigated, it was possible to determine that *H1* can be mitigated using software and redundant sensors, while *H3* can be mitigated using mechanical structures for isolating surfaces that may catch on fire. Comparing the mitigation methods of these two hazards, it is clear that *H1* is a relevant safety hazard to provide evidence for using co-simulation since it is possible to model the sensors, vehicle dynamics, and simulate the software for mitigating the hazard. *H3* is somewhat more difficult to co-simulate since the model would be more focused on the mechanics of the system.

A compact fault-tree analysis of the hazard *H1* is demonstrated in Figure 6a, where the events of the camera view being obscured or a decreased LiDAR performance could potentially lead to *H1* occurring. Possible root causes of these two events are illustrated in the fault-tree as heavy rain and dense fog. This information is used when describing the safety goals that are formalized using GSN, illustrated in Figure 6b. The evidence for fulfilling the safety goals *G2*, *G3* (from Figure 6b) and *G4: The system can tolerate inaccurate sensor measurements* is defined as follows:

Evidence for G2: Co-simulation results of heavy rain scenarios.
Evidence for G3: Co-simulation results of dense fog scenarios.
Evidence for G4: Co-simulation results of scenarios with inaccurate sensor measurements.

5   Producing Evidence using Co-simulation

This section describes the process of creating evidence using co-simulation, by demonstrating how to follow the steps 3, 4, 5, and 6 presented above in Figure 5.

The first step consists of designing test scenarios and creating the relevant models. The test scenarios must be relevant to the physical system and possible to be executed in a co-simulation. The scenarios should be defined clearly in a way that makes it possible to reproduce the results of the co-simulation. They must also be directly related to the safety goals in a GSN, which in some cases are related to the hazard root causes. Table 1 describes the defined test scenarios, related to the GSN in Figure 6b, where the specific hazardous events are defined together with the vehicle speed and distance to the obstacle. These scenarios are typically unethical or expensive to test on the physical system, and therefore can beneficially be co-simulated instead. The goal is to co-simulate the system in these scenarios, and demonstrate its safety under the hazardous conditions.

In order to co-simulate these test scenarios, models of the vehicle dynamics, vehicle controller, sensors, environment and a safety controller must be created.
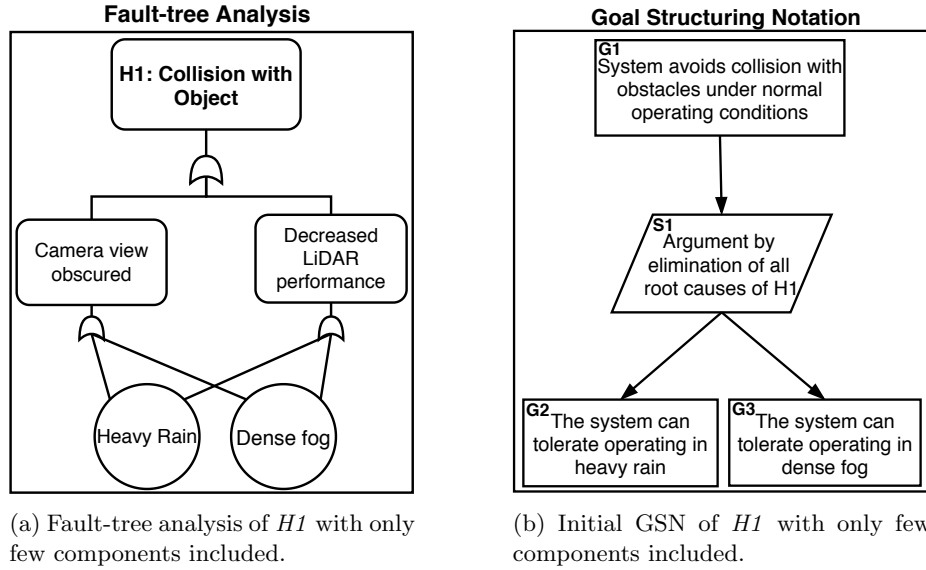
Fault-tree Analysis

(a) Fault-tree analysis of *H1* with only few components included.

Goal Structuring Notation

(b) Initial GSN of *H1* with only few components included.

Fig. 6: Example of how the fault-tree analysis can directly be translated to GSN. Further details of the diagrams can be found in [25].

Hazardous event	Vehicle speed	Initial distance to obstacle
Dense fog	{1,2,3} [m/s]	>1 meter
Heavy rain	{1,2,3} [m/s]	>1 meter
Inaccurate sensor	{1,2,3} [m/s]	>1 meter

Table 1: Defined test scenarios.

Each of the models created in this project are described in Table 2. The FMUs containing the models of the vehicle dynamics and controller were taken from an example project from INTO-CPS[1] [8]. The remaining FMUs were modelled using the python library, PyFMU [19]. The models of the system parts were created with only the necessary details that are relevant in this stage of the project, since a Preliminary safety case is the objective.

The main parts of the process of creating the sensor model are described below. This information is used to illustrate how to combine knowledge from the hazardous events and the sensors of the system. As described above in Section 2.1, a camera, LiDAR and radar are mounted on the system. The effect of dense fog and heavy rain on cameras and state-of-the-art LiDARs is described as

[1] https://github.com/INTO-CPS-Association/example-autonomous_vehicle

[2] https://www.20sim.com/

[3] http://overturetool.org/

[4] https://pypi.org/project/pyfmu/

[5] https://opencv.org/

Model	Program	Description
Vehicle	20-sim[2]	Models the vehicle dynamics using a bicycle model.
Controller	Overture[3]	Controls the vehicle using the pure-pursuit path following algorithm.
Sensor	PyFMU[4]	Models the three types of sensors combined, i.e. LiDAR, radar and camera. A more detailed explanation of this model is presented below.
Environment	PyFMU	Models the obstacles in the environment surrounding the vehicle. The environment was modelled using the python library OpenCV[5]. The environment can be loaded using a white background image, with obstacles defined as coloured objects on the image. This allows easily creating different environments.
Supervisory Controller	PyFMU	Safety controller that performs a safety stop before crashing into an obstacle. The controller performs a safety stop if a detected obstacle is within the range of the calculated braking distance of the vehicle. The obstacles are detected using the sensor model. Therefore, if the sensor model does not detect an obstacle, then the Supervisory Controller will not receive information about an obstacle, and will therefore not perform a safety stop.

Table 2: Models used in the project, together with the program that they were modelled in, and a description of each model.

a reduction in the maximum viewing range of the sensors [2, 12, 20, 17]. Figure 7 illustrates how dense fog and heavy rain can be modelled using a single sensor model at a high abstraction level.

Figure 8 illustrates how the sensor model combines all the sensors into one, instead of creating a model of each sensor. This allowed modelling the complete effect of the environment, using adjustable parameters defining the minimum and maximum sensor range. The minimum range of the sensor represents the hazard of inaccurate sensor measurements [18, 5]. In the future, when more knowledge about the sensors and the system is obtained, details can be added to the constructed models iteratively [21].

In total, 2 scenarios were co-simulated, with the parameters described below, leading to in total 12 co-simulation runs:

Environment: Two different obstacle maps, with varying obstacle locations, densities and sizes.

Vehicle speed: 1, 2, and 3 [m/s]. These values are taken from [13], which describes the optimim combine harvester speed is around 2.2 [m/s].

Minimum sensor range: When operating correctly, the range was modelled as 0.5m. When operating with inaccurate measurements, the range was modelled as 1m [18, 5].
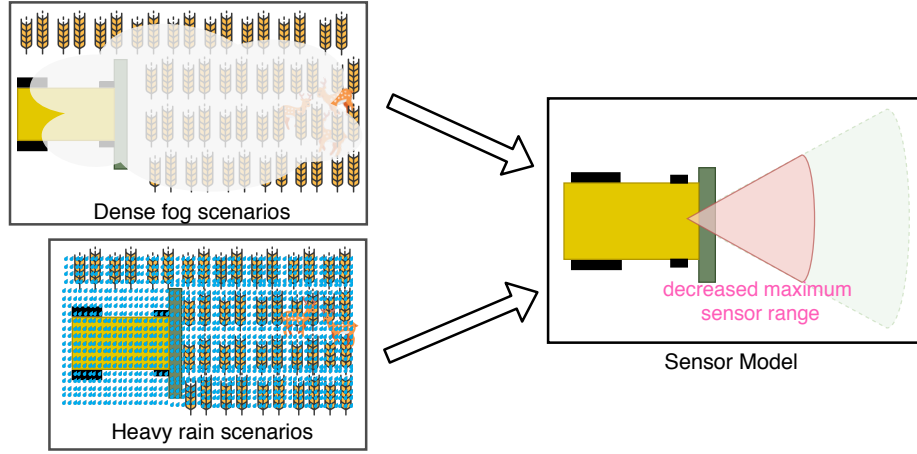
Fig. 7: An illustration of how the scenarios related to dense fog and heavy rain may be modelled simultaneously, since they affect the system in similar ways. The figure demonstrates that the scenarios where dense fog or heavy rain occur affect the maximum viewing range of the sensor.

(a) A mounted lidar and radar on a combine harvester, showing the range of each of the sensors. This is just an example, showing two mounted sensors, but in reality there will be several types of sensors mounted in different places on such a system.

(b) An example illustrating how the sensors are modelled, by generalizing a combination of all of the sensors on a combine harvester. The range of the generalized sensors together is modelled as a circular area around the vehicle, with a minimum and maximum range.
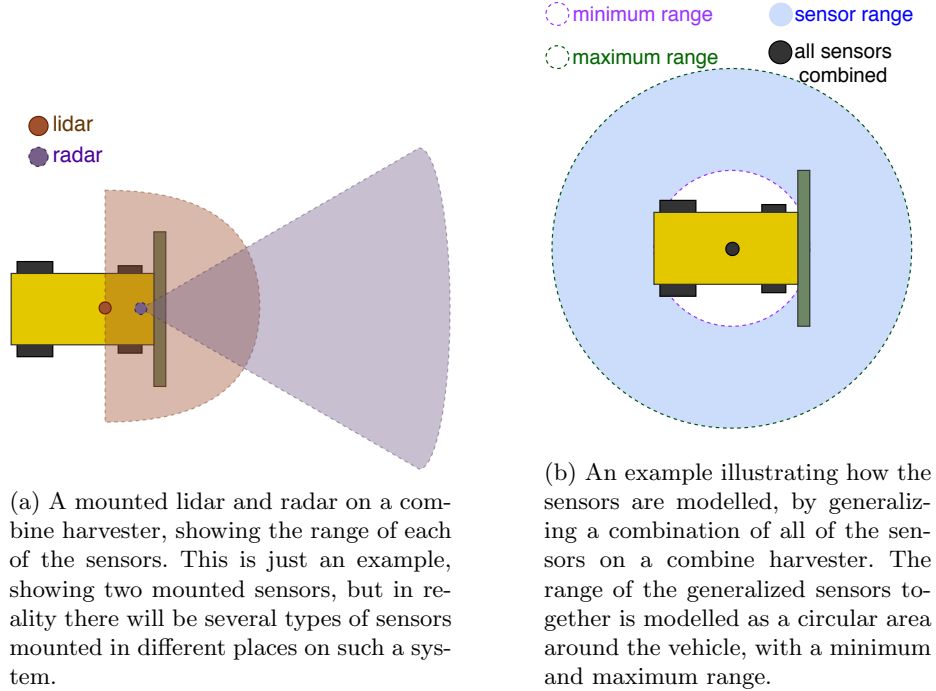
Fig. 8: Generalization of sensors on the autonomous combine harvester.

Maximum sensor range: When operating in dense fog or heavy rain, the range was modelled as 2m, which is the worst case values found in the articles [2, 12, 20, 17].

The different values of the parameters and number of co-simulation runs must be sufficient in order to be able to use the results as evidence. The worst-case values were chosen, but potentially more confidence in the evidence of these hazardous conditions could be obtained using best-case values and at least one value in between.

Figure 9 shows one of the live plots of a co-simulation run, where the vehicle successfully performs a safety stop before colliding with the detected obstacles. In the cases where the vehicle failed to perform a safety stop, the system had run over the obstacles.
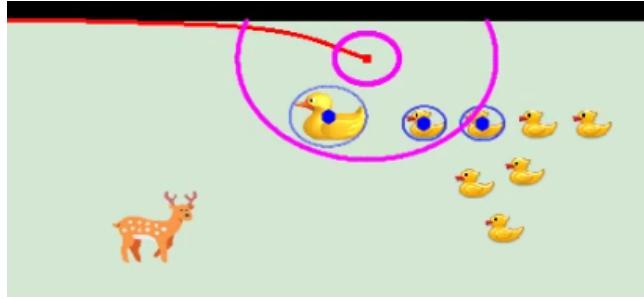

Fig. 9: An example of a co-simulation run of the system with visualization of the minimum and maximum range of the sensor, and the currently detected objects. The red lines displays the route the vehicle has driven so far. The two pink circles represent the sensor minimum and maximum range. The blue circles represent the detected objects. The minimum and maximum ranges are set to 0.5m and 2m respectively in this co-simulation, and the vehicle has performed a safety stop.

The co-simulation results were successful for the scenarios simulating the dense fog and heavy rain, but unfortunately failed in a number of the scenarios simulating the inaccurate sensor measurements. The successful results were used as evidence in the GSN shown in Figure 10; notice the validated model is used as part of the argument for justifying the use of co-simulation as evidence. The model is not yet validated, and therefore the away goal, *G0_ModelValidated*, describing it is greyed out. The unsuccessful co-simulation results mean that improvements on the safety mechanisms must be made, in order to obtain the evidence *Sn3*. It is important to note that the safety properties derived by co-simulation results will only be valid to be used as evidence, if the model of the system is validated and sufficiently represents the true system.
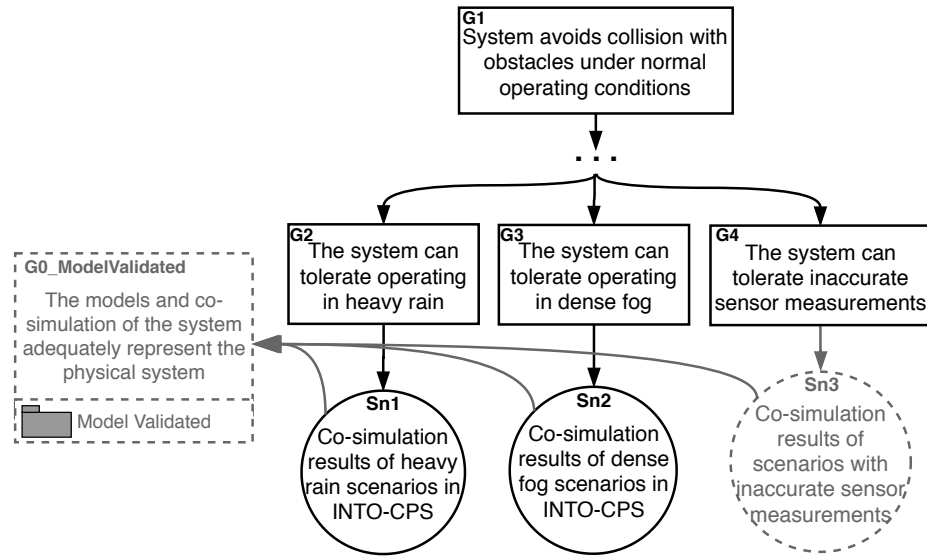
Fig. 10: The relevant parts of the GSN diagram, illustrating which evidence was obtained using co-simulation. The dashed border of *Sn3* and *G0_ModelValidated* illustrates that these components are yet to be constructed, i.e. the evidence for *Sn3* must be obtained, and the model used in the evidence *Sn1* and *Sn2* must be validated against the physical system.

6   Concluding Remarks and Future Work

This paper has demonstrated guidelines for incorporating co-simulation into the process of developing a safety case using FMI-based co-simulation. This was demonstrated by applying the process to a case study of an autonomous combine harvester.

Co-simulation is a strong tool for experimenting with system behaviour, when the system consists of software, electrical and mechanical parts that work together. It is especially relevant to use in the safety case of a complex system, where physical tests of the system are either unethical or challenging to perform. The combine harvester was beneficially co-simulated in scenarios such as dense fog and heavy rain, which are conditions that cannot be controlled in the physical world. The co-simulation was visualized, allowing to create videos of the results, which can advantageously be presented to the regulatory bodies, to help them easier understand how the safety mechanisms ensure system safety under specific conditions.

Different techniques for producing evidence exist, as illustrated in Figure 11, where it is important to consider which technique is most suitable to use depending on the nature of the problem. Looking at co-simulation, the time spent on modeling, co-simulating, analyzing the results and validating the models used, is important to consider. In some cases co-simulation may require too much time, and there may be other techniques that can formally prove the safety of the system, such as a mathematical proof.

This paper has shown how co-simulation can be used to demonstrate that the design of a system has adressed the identified hazards. This was achieved by running co-simulations of a system model in different hazardous conditions, and using the co-simulation results as evidence in the GSN of a safety case.

Future Work

The work presented in this is based on a specific case study with focus on one hazard, and therefore there are multiple potential areas that can be researched:

Extend process: The process could be extended to both interim and operational safety cases, potentially providing guidelines on how the model fidelity can be increased between each phase.

Amount of parameters per scenario: Determining how many parameters should be experimented with is important, to ensure sufficient results for evidence, but also avoid running an endless amount of tests.

Model validation: The model used in this project must be validated, and the methods used to validate it can be explained. Choosing the specific scenarios and parameters to use during validation should be part of this research.

Evaluate process on other systems: Using this process on other types of systems, such as medical devices which require human operators, could demonstrate if the process can directly be transferred to other areas or if adjustments must be made.
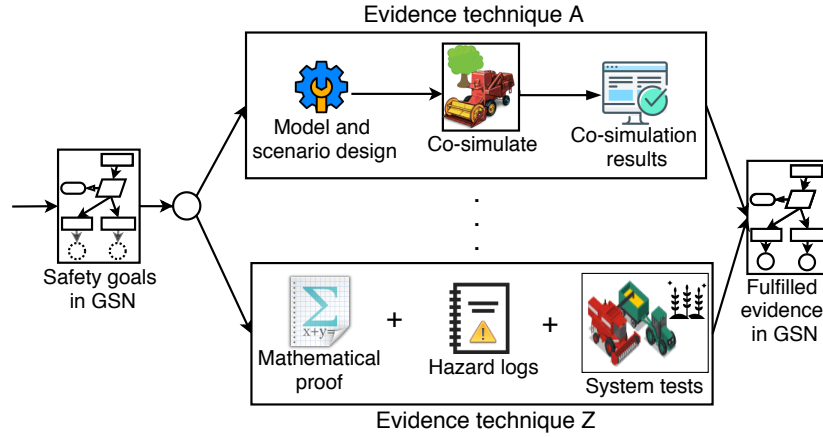
Fig. 11: An illustration of the possible techniques for producing evidence that can be used in a safety case. The specific technique must be determined after defining the safety goals. Note that evidence technique A also requires model validation in order to be able to use the co-simulation results as evidence.

References

1. van Amerongen, J.: Dynamical Systems for Creative Technology. Controllab Products, Enschede, Netherlands (2010)
2. Bijelic, M., Gruber, T., Ritter, W.: A benchmark for lidar sensors in fog: Is detection breaking down? In: 2018 IEEE Intelligent Vehicles Symposium (IV). pp. 760–767 (6 2018). https://doi.org/10.1109/IVS.2018.8500543
3. Bishop, P.G., Bloomfield, R.E.: A methodology for safety case development. In: Safety-critical Systems Symposium. Birmingham, UK (February 1998)
4. Blochwitz, T.: Functional Mock-up Interface for Model Exchange and Co-Simulation. https://www.fmi-standard.org/downloads (July 2014)
5. Clark, R.: Selecting a lidar system, http://www.acuitylidar.com/PDFs/Selecting\%20a\%20LIDAR\%20System.pdf
6. Despotou, G., White, S., Kelly, T., Ryan, M.: Introducing safety cases for health it. In: Breu, R., Hatcliff, J. (eds.) Proceedings of the 4th International Workshop on Software Engineering in Health Care, SEHC 2012, Zurich, Switzerland, June 4-5, 2012. pp. 44–50. IEEE Computer Society (2012)
7. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**(7), 1165–1178 (2008)
8. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In:

FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)

9. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2013)

10. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. ACM Comput. Surv. **51**(3), 49:1–49:33 (May 2018)

11. Group, T.A.C.W.: Goal structuring notation community standard (2 2018), `https://scsc.uk/r141B:1?t=1`

12. Heinzler, R., Schindler, P., Seekircher, J., Ritter, W., Stork, W.: Weather influence and classification with automotive lidar sensors. pp. 1527–1534 (06 2019). https://doi.org/10.1109/IVS.2019.8814205

13. Isaac, N., Quick, G., Birrell, S., Edwards, W., Coers, B.: Combine harvester econometric model with forward speed optimization. Applied Engineering in Agriculture **22** (06 2006). https://doi.org/10.13031/2013.20184

14. Kelly, T.: A systematic approach to safety case management (01 2003). https://doi.org/10.4271/2004-01-1779

15. Kelly, T., Weaver, R.: The goal structuring notation – a safety argument notation. In: Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases (2004)

16. Knight, J.C.: Safety critical systems: Challenges and directions. In: Proceedings of the 24th International Conference on Software Engineering. p. 547–550. Association for Computing Machinery, New York, NY, USA (2002)

17. Koyluoglu, T., Hennicks, L.: Evaluating rain removal image processing solutions for fast and accurate object detection. Master's thesis, KTH ROYAL INSTITUTE OF TECHNOLOGY, Sweden (2019)

18. Lee, S., Cho, H., Yoon, K.J., Lee, J.: Intelligent Autonomous Systems 12: Volume 1 Proceedings of the 12th International Conference IAS-12, Held June 26-29, 2012, Jeju Island, Korea. Springer Publishing Company, Incorporated (2012)

19. Legaard, C.M., Gomes, C., Larsen, P.G., Foldager, F.F.: Rapid Prototyping of Self-Adaptive-Systems using Python Functional Mockup Units. SummerSim '20, ACM New York, NY, USA (2020)

20. Liu, Z., He, Y., Wang, C., Song, R.: Analysis of the influence of foggy weather environment on the detection effect of machine vision obstacles. Sensors **20**(2), 349 (1 2020). https://doi.org/10.3390/s20020349, `http://dx.doi.org/10.3390/s20020349`

21. Maria, A.: Introduction to modeling and simulation. In: Proceedings of the 29th Conference on Winter Simulation. pp. 7–13. IEEE Computer Society (1997)

22. Martin, H., Tschabuschnig, K., Bridal, O., Watzenig, D.: Functional Safety of Automated Driving Systems: Does ISO 26262 Meet the Challenges?, pp. 387–416. Springer International Publishing (September 2017)

23. Sommerville, I.: Software Engineering. Addison-Wesley, 5th edition edn. (1996), iSBN 0-201-42765-6

24. Sun, L., Kelly, T.: Safety arguments in aircraft certification. In: 4th IET International Conference on Systems Safety 2009. Incorporating the SaRS Annual Conference. pp. 1–6 (2009)

25. Tola, D.: A Co-Simulation Based Approach for Developing Safety-Critical Systems. Master's thesis, Aarhus University, Department of Engineering (June 2020)

26. Val, J., Videgain, M., Martin-Ramos, P., Cortés, M., Boné-Garasa, A., Ramos, F.: Fire risks associated with combine harvesters: Analysis of machinery critical points **9**, 877 (12 2019). https://doi.org/10.3390/agronomy9120877