Proceedings of the Workshop on Formal Methods in Human Computer Interaction (FoMHCI) 2015, Duisburg, Germany

Eds.: Benjamin Weyers, Judy Bowen, Alan Dix, Philippe Palanque

FoMHCI Website: https://sites.google.com/site/wsfomchi

In Conjunction with the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS): http://eics2015.org/

Title: Proceedings of the Workshop on Formal Methods for Human Computer Interaction 2015

Editors: Benjamin Weyers, RWTH Aachen University, Germany
Judy Bowen, Waikato University, New Zealand
Alan Dix, Universiy of Birmingham, United Kingdom
Philippe Palanque, Université Paul Sabatier – Toulouse III, France

Corresponding Editor: Benjamin Weyers

RWTH Aachen University
IT Center - Computational Science & Engineering
Computer Science Department - Virtual Reality Group
Jülich Aachen Research Alliance - JARA-HPC

Seffenter Weg 23, 52074 Aachen, Germany
Phone +49 241 80-24920
Email: weyers@vr.rwth-aachen.de
URL: www.vr.rwth-aachen.de

Full text accessible at: urn:nbn:de:hbz:82-RWTH-2015-030425

Publikationsserver

Universitätsbibliothek
RWTH Aachen University
Templergraben 61
52062 Aachen, Germany
www.ub.rwth-aachen.de

Content

Foreword

The Workshop on Formal Methods in Human Computer Interaction (FoMHCI) is meant to bring scientists and interested researchers together who are interested in formal methods in the context of user interfaces, interaction techniques, and interactive systems. Its intention is to present and discuss existing formal methods for the following (not exhaustive) list of topics:

- Description and modeling of user interfaces, interaction techniques, and interactive systems
- Validation and verification concepts and techniques for user interface, interaction techniques and process, as well as interactive systems
- User modeling techniques, concepts, and languages
- Task modeling techniques, concepts, and languages
- Modeling and description techniques for post desktop interaction concepts and metaphors
- Execution and adaptation of formal descriptions
- Description of adaptive interactive systems and user interfaces
- Multi-user and multi-view systems and interaction

Two use cases were provided which could be used for the discussion of the presented formal methods in the submissions. The first describes the partially automated steering of a nuclear power plant, where the second concentrates on a multi-user scenario in context of air traffic control.

The authors were asked to consider the following points if preparing their submissions for the workshop:

- The consideration of the use cases was recommended but not mandatory. All kinds of descriptions of formal methods in human computer interaction were welcome.
- If a use case was considered in a contribution, it did not have to be followed in every detail. The detailed descriptions were given to provide a full image of the individual use case. It was further possible to focus on specific parts of the use cases if reasonable.

The papers were juried by the members of the program committee of the workshop and were chosen according to relevance, quality, and likelihood that they stimulate and contribute to the workshop.

The workshop was held in conjunction with the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS) 2015 in Duisburg, Germany on June 23, http://eics2015.org/. The whole description can be accessed through the workshop's website under https://sites.google.com/site/wsfomchi. The description of the workshop has been further published in the proceedings of EICS 2015 and can be accessed through http://dx.doi.org/10.1145/2774225.2777460.

The Organizing Committee

Benjamin Weyers, Judy Bowen, Alan Dix, Philippe Palanque

Organizing Committee

Benjamin Weyers, RWTH Aachen University, Germany

Judy Bowen, University of Waikato, New Zealand

Alan Dix, University of Birmingham, United Kingdom

Philippe Palanque, Université Paul Sabatier- Toulouse III, France

Program Committee

Judy Bowen, University of Waikato, New Zealand

Antonio Cerone, IMT Institute for Advanced Studies Lucca, Italy

José Creissac Campos, University of Minho, Portugal

Paul Curzon, Queen Mary University of London, UK

Anke Dittmar, University of Rostock, Germany

Alan Dix, University of Birmingham, UK

Michael Harrison, Newcastle University, UK

Chris Johnson, University of Glasgow, UK

Célia Martinie, Université Paul Sabatier, Toulouse, France

Paolo Masci, Queen Mary University of London, UK

Mieke Massink, CNR-ISTI, Italy

Philippe Palanque, Université Paul Sabatier, Toulouse, France

Fabio Paterno, ISTI, Pisa, Italy

Steve Reeves, University of Waikato, New Zealand

Rimvydas Rukšėnas, Queen Mary University of London, UK

Neha Rungta, NASA Ames research center, USA

Benjamin Weyers, Aachen University, Germany

FILL: Formal Description of Executable and Reconfigurable Models of Interactive Systems

Benjamin Weyers
RWTH Aachen University
Aachen, Germany
weyers@vr.rwth-aachen.de

ABSTRACT
This paper presents the Formal Interaction Logic Language (FILL) as modeling approach for the description of user interfaces in an executable way. In the context of the workshop on Formal Methods in Human Computer Interaction, this work presents FILL by first introducing its architectural structure, its visual representation and transformation of reference nets, a special type of Petri nets, and finally discussing FILL in context of two use case proposed by the workshop. Therefore, this work shows how FILL can be used to model automation as part of the user interface model as well as how formal reconfiguration can be used to implement user-based automation given a formal user interface model.

Author Keywords
formal modeling; formal languages; user interface description; interaction logic; human computer interaction; automation.

ACM Classification Keywords
H.5.2. Information interfaces and presentation (e.g., HCI): User Interfaces;

INTRODUCTION
Formal methods have a broad use in human computer interaction research and applications. The major benefits of formal modeling methods are the possibility to apply formal verification and analysis methods, the possibility of execution and the application of formal reconfiguration and adaption mechanisms, and finally their description in machine-readable form. This work presents a formal modeling approach that combines these aspects into one formal description language for modeling user interfaces. It combines a visual description language and an algorithmic transformation to reference nets, a special type of Petri nets, which makes it executable and analyzable. It is called the Formal Interaction-Logic Language (FILL). FILL is based on a layered architecture, which structures a user interface into a physical representation as set of widgets and the interaction logic layer, which describes the processing of events emerging from the physical representation or state changes emitted by the controlled system. This modeling approach is accompanied with a modeling, execution and reconfiguration software tool, the UIEditor. It offers visual editors for the creation of FILL-based user interface models, a runtime environment for its execution, as well as tools for the interactive definition and application of reconfiguration operations. The last differentiates FILL from works using Petri nets-based descriptions, such as the Interactive Cooperative Objects described by Navarre et al. [14] or works like these from de Rosis et al. [12] or Janssen et al. [13].

The paper is structured as follows. The next section presents FILL's architectural basis and relates this approach to other existing architectural concepts. It is followed by an introduction of FILL's syntax and semantics as well as a description of the transformation to reference nets. This is followed by the introduction of the formal reconfiguration of these transformed models using a category-based graph rewriting approach. Finally, FILL will be presented by means of a use case addressing automation for a simplified simulation of a nuclear power plant. This use case will present FILL's potentials for the modeling of automation as part of a FILL-based interaction logic description as well as the embedding of automation into interaction logic through interactive reconfiguration of an initial user interface model. The paper will be concluded with a short outlook on future work.

ARCHITECTURE
The Formal Interaction Logic Language (FILL) is a formal language suitable for the description of executable and reconfigurable models of interactive systems. According to the classification presented by Dix in [1], FILL is a description language for the specification of executable system models. It follows a basic three layer architecture, which is conceptually related to the model-view-controller (MVC) pattern [2]. Fig. 1 shows the comparison of the classic MVC pattern and a slightly restructured version (in the middle of Fig. 1), which takes widgets as functional elements of a user interface into account. Here, the controller directly associated to the view of a widget holds the functionality which triggers changes of the visual appearance of the widget that and its state, which is encapsulated as widget model. The widget controller further encapsulates all

To cite this article use the following information:

Benjamin Weyers. 2015. FILL: Formal Description of Executable and Reconfigurable Models of Interactive Systems. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 1-6.
http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

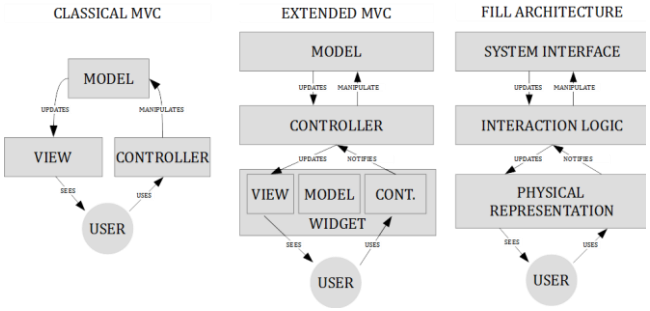

Figure 1. Relation of FILL architecture to an extended version of the MVC software pattern.

functionality that is necessary to generate events as data objects according to user interaction, e.g., by pressing a button. Thus, a widget can be modeled in the sense of a MVC. Nevertheless, a user interface can be assumed being composed by more than one widget, which means that the MVC on widget level exists *n* times for *n* widgets of one user interface. This set of widgets mapped to the physical representation in context of FILL. The non-widget related (global) controller (as shown in Fig. 1) comprises all functionality relevant to update the (global) model representing the state of the controlled system and processes events emitted from the widgets (controller). This global controller refers to the interaction logic in FILL's architecture, where the global model can be associated to the system interface (cf. Fig. 1 right). FILL as language addresses the modeling of the interaction logic, where in the following the combination of a physical representation description and the interaction logic model is referred to as user interface model.

This architecture specifies a macroscopic, horizontal and layered representation of an interactive system, which does not reflect the more fine-grained inner structure of the various layers. “Zooming in” on single widgets and their associated part of interaction logic of a user interface (as defined above) obtains a similar modeling structure as presented by Paternó in [3,4], which is indicated in Fig.1 on the right. These so called *interactors* (see Fig. 2) as modeling concept for interaction objects offer not only the description of widgets including a specific visual representation but further offer the modeling of multi-layer networks of interactors by hierarchical combination of input and outputs of interactors. Every interactor is meant to be related to a specific (elementary) task the user wants to work on with the given interaction system, which is also one of the main aspects of FILL.

As can be seen in Fig. 2, an interactor is composed by three main elements: (a) the abstraction, which contains the description of data which should be visualized, (b) the input, which performs the processing of the input from to user to be redirected to the application or influence the third component (c) the representation, which defines the appearance of the interaction object to the user. Using FILL, the interaction logic of a user interface is separated into so-called *components* or *interaction processes*, which comprise structures that represent elements (a) and (b) of an interactor, where (c) is solely part of the widget the physical representation is composed of.

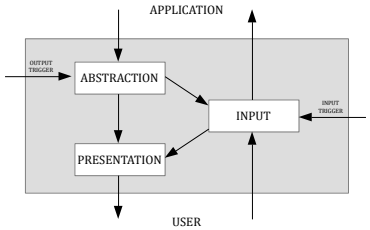

Figure 2. Interactor according to [4], Figure 1.

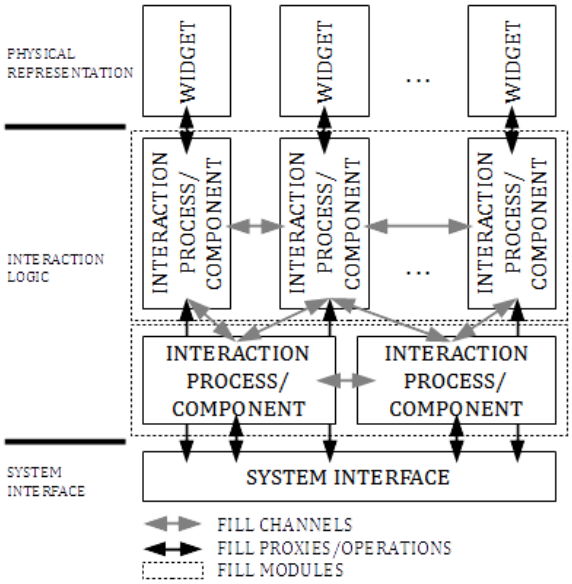

Figure 3. FILL Architecture

Fig. 3 shows the architecture FILL is following. As mentioned above, the structure is oriented along previous work and aims at picking up successful approaches. Therefore, the three layer concept follows the current implementation concepts of GUI toolkits, such as Swing or Qt. Furthermore, according to modern software development, FILL offers mechanisms to structure interaction logic for enabling reusability and successful modeling: A set of components can be further comprised as *module*, which combine components with similar functional semantics, e.g., components that model the communication of widgets with the system to be controlled, as interactors intend to do, or components that implement adaptions of subsets of widgets, which would be described as multi-layered interactors in terms of the architectural approach described by Paternó [3,4]. For the communication with other layers (physical representation or the system interface) or for communication between interaction processes and modules, FILL offers various structures, such as operations and proxies as well as channels. Therefore, the following section will present FILL as modeling language in more detail.

FORMALIZATION

FILL is a formally defined graph-based visual modeling language, whose formal semantics is specified by an algorithmic transformation to a specific type of colored Petri nets, so called Reference Nets [6]. FILL graphs describe processes which model the processing of data items, comparable to process modeling languages, such as the Business Process Modeling Notation (BPMN) [10]. Therefore, data items are passed from node to node activating the function associated with a node every time a data item reaches it.

The visual notation of FILL is shown in Fig. 4. The complete formal definition of FILL can be found under http://www.uiedtior.org/fill.pdf and will not be further discussed here. A detailed description of FILL and its transformation can be found in [9, 11].

FILL is composed of four types of nodes and two types of edges. The first set of nodes contains so-called *operation nodes*. These nodes represent operations, which process data items or redirect to other components, such as the system interface. System operation nodes define connections to or from the system to be controlled and thereby define connections to and from the system interface. Interaction-logic operations define data processing operations, such as arithmetic operations, type conversions or data transformations like the generation of arrays. *Channel operations* define connections to or from other interaction processes and modules in the interaction logic. An input channel operation defines an entry to a channel where an output channel operation defines an output. All operation nodes are comprised of an *input* and/or *output port(s)*, which define the interface of operations. The second set of nodes are the so called *proxy nodes*. Proxy nodes represent the connection to and from a widget. They emit events

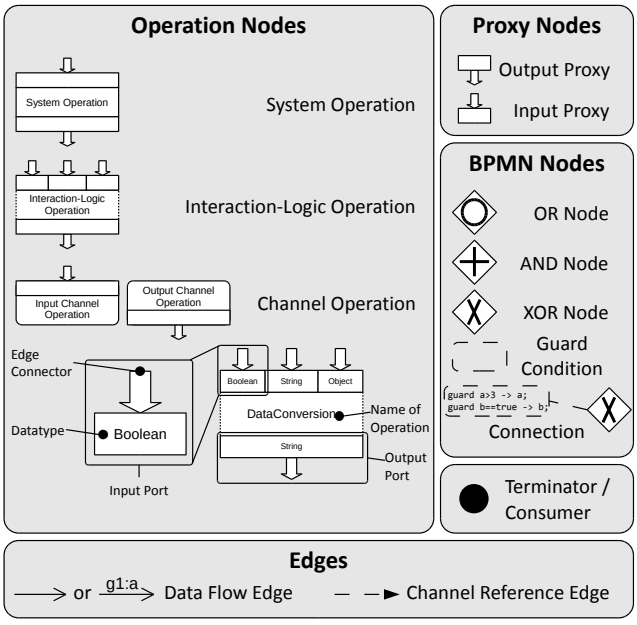

Figure 4. Visual notation of FILL.

emerging from the widget into the FILL interaction process or trigger state changes of the widget emerging from the interaction process. The third set of nodes is dedicated to branching and merging of FILL processes, the so called *BPMN nodes*. These nodes are borrowed from the BPMN notation for the modeling of business processes [10]. There are three types of nodes: the OR, the AND, and the XOR node. These nodes mainly differ in the branching or fusing semantics, which is further specified by a guard condition, which is inscribed to the node. The nodes have either *one* incoming connection and *n* outgoing (branching) or *n* incoming and *one* outgoing connection (fusion). The final set consists only one node representing a *terminator* for FILL processes. This node simply consumes all incoming data items.

All nodes are connected via edges. Two types of edges are distinguished in FILL: *data flow edges* and channel reference edges. Data flow edges specify data flow between nodes. Specifically, data flow edges connect ports, proxies, BPMN nodes and terminators. Channel reference edges are used to associate input to output channel operations over FILL component boundaries. An exemplary FILL model will be discussed in the next section accompanied with the presentation of FILL graphs into reference nets.

Transformation to Reference Nets

As mentioned in the introduction, FILL is a description language for executable system models. For the execution as well as for the definition of formal semantics, a modeled FILL graph is transformed into a reference net representation. The transformation algorithm will be presented by means of a simple example. The whole algorithm has been described in [9].

A FILL-based interaction logic description is transferred into a reference net. Reference nets are a special type of colored Petri net [6]. The main difference to classic colored Petri nets is the feature of references which enables the simulation of a reference net to generate net instance of a net "class". These net instances can reference to each other or to other objects (e.g. JAVA objects) through synchronously linked transitions. Both concepts are used for the transformation of FILL-based interaction logic. An example for such a transformation is shown in Fig. 5.

The interaction process on the left of Fig. 5 shows a process that could, e.g., be triggered by an event emerging from a button that has been pressed. The generated event object is passed into the process through the output proxy node (upper left of the image). This redirects the data item to a system operation entitled "setSV2Status". This operation returns the current value for the variable SV2 as data item into the process, which is in this case a Boolean value. The XOR BPMN node evaluates this data item according to the indicated guard condition. In case it is false, the left branch inscribed with 'a' will be triggered. Otherwise, the branch inscribed with 'b' will be triggered. In both cases, an interaction-logic operation is triggered, which generates a Bool-

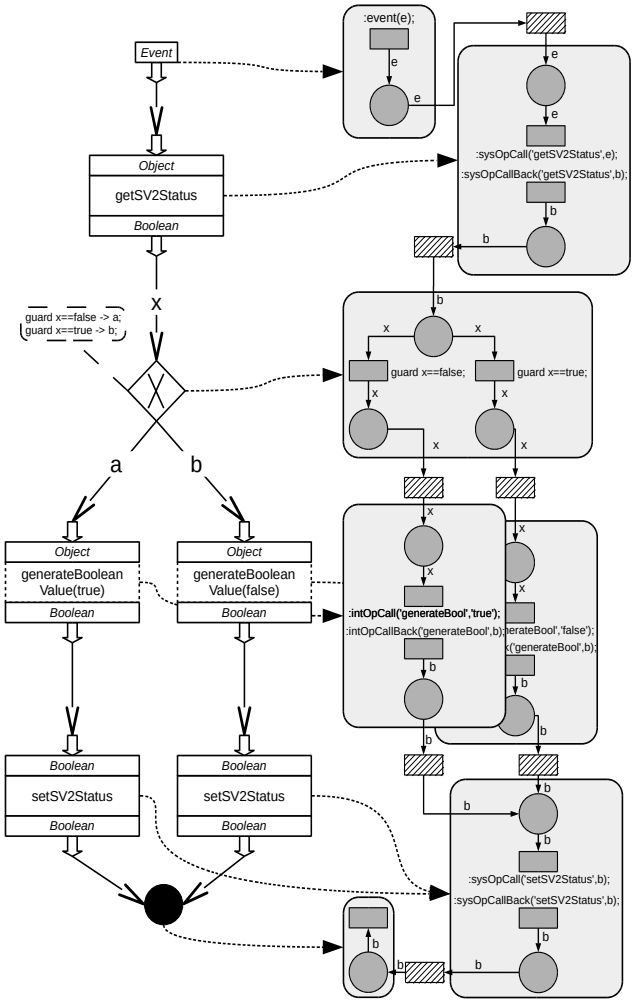

Figure 5. Transformation example user interface before interactive reconfiguration

ean of contrary value. This value is then redirected to a system operation, which sets this value to the system variable SV2. Finally, the process is terminated. In conclusion, the process toggles the value of the system variable SV2 between true and false. The reference net resulting from the transformation is shown on the right of Fig. 5. For the following description, a basic understanding of colored Petri nets is assumed.

For the firing of a transition, a token has to be present that can be mapped to the inscribed variables of incoming edges as well as the inscription of the transition. This is also true for guard conditions as used for the transformation of the XOR node (see Fig. 5. middle right). In the example transformation also the use of synchronous channels in references nets can be seen. Consider for example the transformation of the system operation 'getSV2Status'. Incoming data items are passed to an incoming place where it is consumed by the transition inscribed with ':sysOpCall('getSV2Status',e)'. This synchronous call refers to a channel of name 'sysOpCall' with two parameters. This can be another transition or a function call in a referred application. The latter is possible due to a the simulator provided for reference nets called Renew [5], which will be described in more detail. In this case, sysOpCall refers to a function that is executed on the system interface. After returning from this function, the system interface calls the transition inscribed with ':sysOpCallBack('getSV2Status', b)', where b holds the current value of the system variable SV2, which is transferred to the outgoing place. As mentioned before, further details for the execution of FILL-based interaction logic will be presented in the next section presenting two use cases and the modeling, execution and reconfiguration framework called UIEditor.

Formal Reconfiguration of Interaction Logic

Accompanied with the development of FILL, we further developed a formal reconfiguration concept based on a graph rewriting method. Therefore, transformed FILL models (i.e. reference nets) are rewritten using a concept called Double Pushout Approach (DPO). This concept which is well known in category theory has been first applied to Petri nets by Ehrig et al. [7] and has been extended by us to colored Petri nets, which mainly includes the rewriting of inscribed Petri nets [8].

The major challenge in this context is the algorithmic generation of DPO rules based on a given transformed interaction logic. A basic set of algorithms for the interactive generation of rewriting rules has been implemented in the UIEditor tool, which will be presented in context of the use cases, below. A first approach towards a generic concept for rule generation has been proposed in [9] but will not be further elaborated in this work. The following section will focus on a use case addressing automation in the control of a simplified simulation of a nuclear power plant and will show the interactive creation of reconfiguration rules and their application to interaction logic.

USE CASE

This section presents the visual editor and execution framework for FILL-based user interfaces: the UIEditor. This introduction will present a (partial) automation of a simplified simulation of a nuclear power plant.

UIEditor

The UIEditor is a software for the creation, the execution and the reconfiguration of user interfaces based on a FILL-based description of interaction logic. It is comprised of various visual editors for the creation of the physical representation of a user interface as well as the FILL-based interaction logic using its visual representation to create interaction processes and FILL modules. It includes the implementation for the transformation of FILL models to reference nets as well as a software interface to Renew, which enables the simulation of the transformed reference net for execution. Furthermore, it implements the DPO approach-based reconfiguration of interaction logic as well as an associate visual interface for the interactive reconfiguration of the user interface, as will be described in more detail in Use Case 2, below. Use Case 1 will focus on the creation

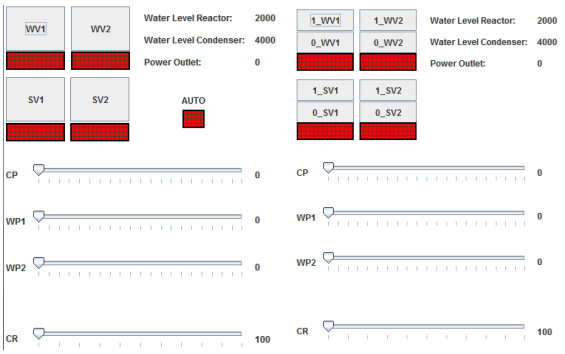

Figure 6: Physical representations of modeled user interfaces in the two presented use cases. Left: with implemented automation in the interaction logic, right: the original user interface before interactive reconfiguration

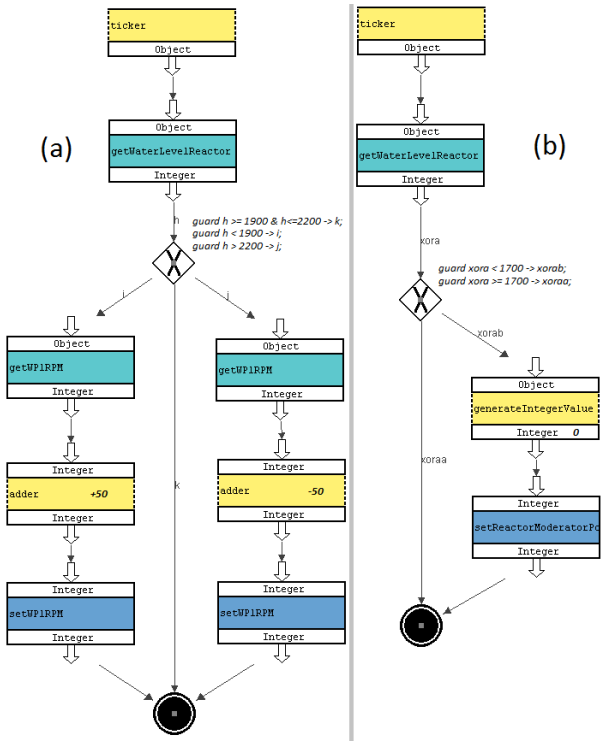

Figure 7: Physical representations of modeled user interfaces in the two presented use cases.

and execution of a user interface which implements a certain type of automation as part of the interaction logic. Both use cases will be presented during the workshop in a live demo using the UIEditor.

Use Case 1: Automation as part of the interaction logic

The central argument for the embedding of automation into the interaction logic of a user interface is that this type of solution makes automation flexible and in case of using a formal method, also verifiable. Using FILL, automation is further reusable by encapsulating the relevant part into modules. Furthermore, automation becomes adaptable using the reconfiguration mechanism. The latter will be further used in the next section to let the user specify certain types of automation.

Here, we will concentrate solely on the description of single aspects of the user interface model and assume that the complete model of the user interface exists as FILL description. In Fig. 6, the modeled physical representation is shown, which offers all necessary widgets to control the simplified simulation of a nuclear power plant as it has been provided. For opening and closing of valves, buttons are provided, sliders offer the manipulation of the pump's speed as well as the position of the control rods. Furthermore, various labels show the current value of the relevant system components.

Fig. 7 shows two modeled examples of possible automation functionality introduced into the interaction logic. Fig. 7 (a) shows a process that controls the speed of water pump 1 (WP1) and is associated with the lamp inscribed with "AUTO". This automation increases the speed of the pump in case the water level drops below 1900mm and decreases it when the water level goes over 2200mm. This process as well as the one presented next is triggered by the "ticker" operation, which periodically triggers an event and generates a simple object to be sent into the interaction process. Fig. 7 (b) shows the implementation of the security shut down (SCRAM) of the reactor which kicks in if the water level of the reactor drops under 1700 mm. In this case, the control rods are inserted completely into the reactor core.

The visual representation of the FILL processes are taken from the UIEditor as screenshots. Here, the predefined values associated with some of the interaction-logic operations as well as the guard conditions for the BPMN nodes were additionally annotated because they are not visible in the graph representation but can be entered and changed through a context menu popping up by double-clicking on the node in the graph.

Use Case 2: User-driven Automation

As mentioned above, using the UIEditor the reference net-based representation of a FILL model can be reconfigured using a formal graph rewriting approach. Therefore, the UIEditor offers a visual interface for the creation of reconfiguration rules selecting widgets in the physical representation and by applying certain reconfigurations to the associated interaction process. This reconfiguration adds new parts to the interaction logic as well as new widgets to the physical representations. For this use case of a user-driven automation, we want to present two reconfiguration operations: First, for the *fusion* of a given set of interaction processes, which generates a widget that triggers all interaction processes of the previously selected ones. Second, the *discretization* of widgets. It can be applied to a widget that generate values out of a range, such as a slider, and it generates a widget which generates a single value, such as a button. Fig. 6 right shows a slightly changed user interface compared to that shown in Fig. 6 on the left. This is due to the change of buttons from ones which toggle the status of a

valve to a set of widget that can open or close a specific valve. This makes it easier for the following description to generate relevant automations.

The goal is to create a button to SCRAM the reactor. To do so, WP1 should run with 800 rpm, the control rods should be completely inserted into the core and the valves WV1 and SV2 should be open, while the other valves are closed. Furthermore, the condenser pump should run with 1600 rpm. In a first step, according to the two pumps and the control rods, discrete values are selected and related to three new buttons, as shown in Fig. 8 on the right. These are then fused together with the buttons for opening WV1 and SV2 as well as the closing buttons for WV2 and SV1 to a final button labeled "SCRAM" as can be seen in Fig, 8 in the middle.

Further reconfiguration operations are currently implemented, such as *duplicate* widgets, *replace widgets* for output, or fusion of widgets with respect to execute the related operations sequentially (*sequential fusion*) instead of in parallel as done by the previously introduced fusing operation.

CONCLUSION AND FUTURE WORK

This work presented FILL as model approach for the formal description and specification of interaction logic. FILL's definition, its reconfiguration as well as its associated modeling tool, the UIEditor, have been introduced. Finally, FILL has been exemplarily presented by means of a partial automation of a simplified simulation of a nuclear power plant.

Future work mainly addresses the extension of FILL to multi-device description, which further includes mobile devices as well as devices, which do not offer a graphical user interface, instead relying on gesture or audible interaction. Furthermore, we intend to extend FILL with operations that encapsulate reference nets modeled by the user. This can further reduce the use of references to code as is currently the case for the implementation of interaction-logic operations. Finally, the rule generation for the application of reconfiguration to a given formal user interface model has to be investigated in more detail.

REFERENCES

1. Dix, Alan J. (2014): Formal Methods. In: Soegaard, Mads and Dam, Rikke Friis (eds.). "The Encyclopedia of Human-Computer Interaction, 2nd Ed.". Aarhus, Denmark: The Interaction Design Foundation
2. Hartson, R., Gray, P., Temporal Aspects of Tasks in the User Action Notation, Human Computer Interaction, 7, 1-45.
3. Paterno, F., Leonardi, A., A Semantic-based Approach for the Design and Implementation of Interaction Objects. Eurographics 94, 13(3).
4. Paterno, F., A Theory of User-interaction Objects, J. of Visual Languages and Computing (1994) 5, 227-249.

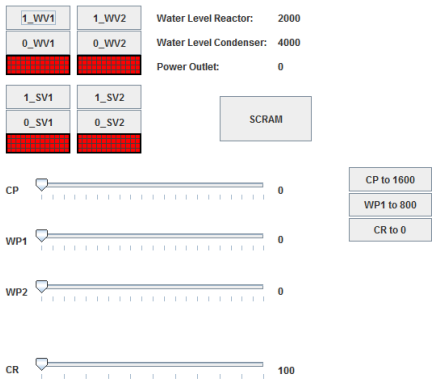

Figure 8: Reconfigured user interface

5. Kummer, O., Wienberg, F., Duvigneau, M., Köhler, M., Moldt, D., and Rölke, H. Renew–the reference net workshop. Proc. of 21st International Conference on Application and Theory of Petri Nets, pp. 87-89. 2000.
6. Kummer, O. Referenznetze. Dissertation. Logos, 2002.
7. Ehrig, H., Hoffmann, K., and Padberg, J.. Transformations of Petri nets. Electronic notes in theoretical computer science 148.1 (2006): 151-172.
8. Stückrath, J., Weyers, B., Lattice-extended Coloured Petri Net Rewriting for Adaptable User Interface Models, Electronic Communications of the EASST 2014, 67.
9. Weyers, B., Borisov, N., Luther, W., Creation of Adaptive User Interfaces through Reconfiguration of User Interface Models using and Algorithmic Rule Generation Approach, Int. Journal On Advances in Intelligent Systems, 7(1&2), 302-325, 2014.
10. White, S. A. BPMN modeling and reference guide: understanding and using BPMN. Future Strategies, 2008.
11. Weyers, B., Burkolter, D., Luther, W., and Kluge, A. (2012). Formal modeling and reconfiguration of user interfaces for reduction of errors in failure handling of complex systems. Int. Journal of Human-Computer Interaction, 28(10), 646-665.
12. de Rosis, F., Pizzutilo, S., and De Carolis, B. Formal description and evaluation of user-adapted interfaces. International Journal of Human-Computer Studies 49, 2 (1998), 95–120.
13. Janssen, C., Weisbecker, A., and Ziegler, J. Generating user interfaces from data models and dialogue net specifications. In Proc. of the INTERACT'93 and CHI'93, 1993, 418–423.
14. Navarre, D., Palanque, P., Ladry, J.-F., and Barboni, E. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Transactions on Computer-Human Interaction 16, 4 (2009), 1–18.

Tasks Decomposition of System Models for Human-Machine Interaction Analysis

Guillaume Maudoux
Université catholique de Louvain, Belgium
guillaume.maudoux@uclouvain.be

Sébastien Combéfis
École Centrale des Arts et Métiers, Belgium
s.combefis@ecam.be

Charles Pecheur
Université catholique de Louvain, Belgium
charles.pecheur@uclouvain.be

ABSTRACT

This paper is concerned with the problem of learning how to interact safely with complex automated systems. With large systems, human-machine interaction errors like automation surprises are more likely to happen. Previous works have introduced the notion of full-control mental models for operators. These are formal system abstractions embedding the required information to control a system completely and without surprises. Full-control mental models can be used as training material but are ineffective as their control over a system is only guaranteed when fully learned.

This work investigates the problem of decomposing full-control mental models into smaller independent tasks. These tasks each allow to control a subset of the system and can be learned incrementally to control more and more features of the system. This paper proposes an operator that describes how two mental models are merged when learned sequentially. With that operator, we show how to generate a set of small tasks with the required properties.

ACM Classification Keywords

B.8.2. Performance and Reliability: Performance Analysis and Design Aids; D.2.4. Software/Program Verification: Formal methods; H.1.2. Models and Principles: User/Machine Systems

Author Keywords

Task decomposition; LTS; Task analysis.

INTRODUCTION

The field of human-computer interaction analysis formalises how human operators interact with automated systems and studies how to assert and improve the quality of these interactions. An important problem is to ensure that humans can interact with a system without surprises and provide a description of such interactions.

Surprises are defined as mismatches between expectations of the operator and the actual behaviour of a system. An operator maintains a model of the system called a mental model[7].

To cite this article use the following information:

Guillaume Maudoux, Sébastien Combéfis, and Charles Pecheur. 2015. Tasks Decomposition of System Models for Human-Machine Interaction Analysis. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 7-12.
http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

He builds it by experimenting on and learning about the system. Operators are assumed to behave according to their mental model. Therefore, mental models should always allow to control the system in use. In that context, learning new features of the system should be done in such a way that the new mental model of the users also allow to control the system. In particular, operators must not interact with the system until the end of a learning phase and cannot fill their functions during that time.

Operators that have learnt all the possible behaviours of a system have built a full-control mental model. Such models have been defined in [4] and techniques to build minimal ones have been described in [3] and [2]. These models allow to control safely all the features of a system. However, learning full-control mental-models is impractical as it implies to learn all the features of a system in one big step. This means that newly hired operators are useless before they master the full complexity of the system. Large systems might even be too complex for one operator to manage. In that case, the system must be split in tasks dedicated to different operators.

To be practical, learning processes should provide a set of tasks in the form of small compatible mental models that can be combined incrementally into bigger models. Each task and each intermediate mental model should ensure safe interactions with the system without necessarily describing all of its features.

Our work investigates how to decompose a full-control mental model into smaller tasks that individually control the system. Learned sequentially, these tasks should augment the mental model of the operator until he possesses a full-control mental model of the system.

Tasks have long been used in the context of human-machine interactions. They can be defined during the system design process and used for system validation like in [5]. They can also be synthesised through a goal that a user needs to achieve and relate to controler synthesis as explained in the Ramadge-Wonham framework [6]. In this work, we propose tasks that have the property of being small and combinable, with no guarantee that they correspond to meaningful objectives for users.

In this paper, we introduce an operator to combine mental models and we argue that it is coherent with the intuition of learning a task. We describe the related decomposition operation and show that mental models can be decomposed into a finite set of basic mental models. We also show that it is pos-

sible to build a set of tasks that properly control the system and such that all their combinations also control the system and eventually have full-control over it. Finally, we define the task complexity as a measure of the difficulty to learn a system.

The remaining of this article is organized as follows. First we introduce the required definitions of mental models and controllability in the"background" section. In the "HMI-LTS decomposition" section, we introduce the composition operator and the decomposition properties. Finally, we show how to obtain the desired decomposition in the section "Full-control mental model decomposition".

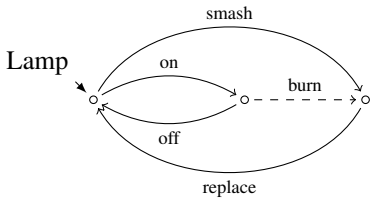

Figure 1. An HMI-LTSs model of a lamp with four commands and one observation. A lamp can be switched on and off as long as it does not burn. When burned or smashed, the lamp needs to be replaced and we are back to the starting point. This HMI-LTS is our its simplicity, it is also its own and only minimal full-contol mental model.

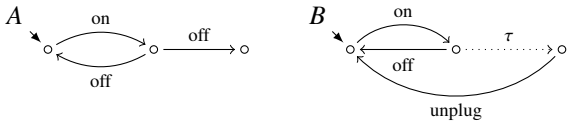

Figure 2. Two examples of nondeterministic systems. *A* **can be turned on then off at least once, but it is impossible to say if it can be turned on again.** *B* **can be turned on and off, but it can also unobsevably change to a state where the only way to restart it is to unplug it.**

BACKGROUND

In this section we define HMI-LTSs, mental models and the full control property. We also introduce full-control mental models, a concept that lies at the intersection of these three notions. This section is intended as a reminder of the required concepts defined in [1].

We start with the concept of labelled transition systems for human-machine interactions (HMI-LTSs) which are slightly modified labelled transition systems (LTSs). An LTS is a state transition system where each transition has an action label. LTSs interact with their environment based on this set of actions. Additionally, LTSs can have an internal τ action that cannot be observed by the environment. Two small LTSs are shown in figure 2.

DEFINITION 1 (LABELLED TRANSITION SYSTEM). *A labelled transition system (LTS) is a tuple $\langle S, \mathcal{L}, s_0, \rightarrow\rangle$ where S is a finite set of states, $\mathcal{L}$ is a finite set of labels representing visible actions, $s_0 \in S$ is the initial state and $\rightarrow \subseteq S \times (\mathcal{L} \cup \{\tau\}) \times S$ is the transition relation, where $\tau \notin \mathcal{L}$ is the label for the internal action.*

The executions of LTSs can be observed from the environment via traces. An *execution* of an LTS is a sequence of transitions $s_0 \xrightarrow{a_1} s_1 \dots s_{n-1} \xrightarrow{a_n} s_n$ where each $(s_{i-1}, a_i, s_i) \in \rightarrow$. A *trace* of an LTS is a sequence $\sigma = a_1, a_2, \dots a_n$ where each $a_i \in \mathcal{L}$ and such that there exists an execution $s_0 \xrightarrow{\tau* a_1 \tau*} s_1 \dots s_{n-1} \xrightarrow{\tau* a_n \tau*} s_n$, where $s_0 \xrightarrow{\tau* a_1 \tau*} s_1$ is itself an execution whose only observable action is a_1. For example the Lamp system of figure 1 can exhibit the trace "on, off, on, off, burn" and the trace "smash, replace, on" among infinitely many other.

HMI-LTSs refine LTS by distinguishing two kinds of actions, *commands* and *observations*. Like any I/O transition system, observations are uncontrollable outputs generated by the system and commands are controllable inputs. HMI-LTSs are exactly equivalent to Tretmans' LTS/IOs[8].

DEFINITION 2 (HUMAN-MACHINE INTERACTION LTS). *A human-machine interaction labelled transition system (HMI-LTS) is a tuple $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow\rangle$ where $\langle S, \mathcal{L}^c \cup \mathcal{L}^o, s_0, \rightarrow\rangle$ is a labelled transition system, $\mathcal{L}^c$ is a finite set of command labels and $\mathcal{L}^o$ is a finite set of observation labels. The two sets $\mathcal{L}^c$ and $\mathcal{L}^o$ are disjoint and the set of visible actions is $\mathcal{L} = \mathcal{L}^c \cup \mathcal{L}^o$.*

HMI-LTSs are used to describe both systems and mental models. Mental models are user views of a system. They are by definition deterministic and represent the knowledge an operator has about the system he controls. It is important to note that mental models do not represent the behaviour of a user, but the behaviour of a system as seen by a user. A command in a mental model corresponds exactly to the same command on the system. The interactions between a system $\mathcal{S}$ and an operator behaving according to its mental model $\mathcal{M}$ are defined by the synchronous parallel composition $\mathcal{S} \parallel \mathcal{M}$. This distinguishes HMI-LTSs from LTS/IOs where inputs of the system must be synchronised on the outputs of the user and vice versa.

In addition, we want mental models to control systems without surprises. In particular, we want to avoid mental models to contain commands that are impossible on the system and to ignore observations that the system could produce. This motivates the introduction of the control property.

The following definition uses the s **after** σ operator that describes the set of states that can be reached from the state s after an execution whose observable trace is σ. Also, $A^c(s)$ (resp. $A^o(s)$) is the set of possible commands (resp. observations) of s. An action is possible in s if it is the first action of some trace starting at s. Moreover, an LTS is deterministic if $|s \textbf{ after } \sigma| \leq 1$ for any σ. The HMI-LTS A from figure 2 can be in two states after the trace "on, off" and is therefore not deterministic. The HMI-LTS B has two possible actions in its middle state: 'off' and 'unplug'

DEFINITION 3 (CONTROL PROPERTY). *Given two HMI-LTSs $\mathcal{S} = \langle S_{\mathcal{S}}, \mathcal{L}^c, \mathcal{L}^o, s_{0\mathcal{S}}, \rightarrow_{\mathcal{S}}\rangle$ and $\mathcal{M} = \langle S_{\mathcal{M}}, \mathcal{L}^c, \mathcal{L}^o, s_{0\mathcal{M}}, \rightarrow_{\mathcal{M}}\rangle$, $\mathcal{M}$ controls $\mathcal{S}$ if $\mathcal{M}$ is deterministic and for all traces $\sigma \in \mathcal{L}^*$ such that $s_{\mathcal{S}} \in s_{0\mathcal{S}}$ **after** σ and*

$\{s_{\mathcal{M}}\} = s_{0\mathcal{M}}$ **after** σ *:*

$$A^c(s_{\mathcal{S}}) \supseteq A^c(s_{\mathcal{M}}) \text{ and } A^o(s_{\mathcal{S}}) \subseteq A^o(s_{\mathcal{M}}).$$

This definition is symmetric because it allows the mental model not to know the full set of available commands while allowing the system to produce less observations than expected by the mental model. From now on, this is the formal definition we refer to when we say that a mental model controls a system.

For a given system, there always exists a mental model that contains no commands and still allows to control the system. That mental model contains only the traces of observations available from the initial state and corresponds to the mental model needed by an agent to avoid surprises when not interacting with a system. For example, you need to know that your desk phone may ring even when you do not want to interact with it. Someone who ignores that fact will be surprised whenever the phone rings.

We see that a mental model that controls a system does not necessarily explore the full range of possible behaviours of that system. When a mental-model ensures control over a system and allows to access all the available commands of the system, we say that the model fully controls the system.

DEFINITION 4 (FULL-CONTROL PROPERTY). *Given two HMI-LTSs* $\mathcal{S} = \langle S_{\mathcal{S}}, \mathcal{L}^c, \mathcal{L}^o, s_{0\mathcal{S}}, \rightarrow_{\mathcal{S}} \rangle$ *and* $\mathcal{M} = \langle S_{\mathcal{M}}, \mathcal{L}^c, \mathcal{L}^o, s_{0\mathcal{M}}, \rightarrow_{\mathcal{M}} \rangle$*,* $\mathcal{M}$ *is a full-control mental model for* $\mathcal{S}$*, which is denoted* $\mathcal{M}$ **fc** $\mathcal{S}$*, if* $\mathcal{M}$ *is deterministic and for all traces* $\sigma \in \mathcal{L}^*$ *such that* $s_{\mathcal{S}} \in s_{0\mathcal{S}}$ **after** σ *and* $\{s_{\mathcal{M}}\} = s_{0\mathcal{M}}$ **after** σ *:*

$$A^c(s_{\mathcal{S}}) = A^c(s_{\mathcal{M}}) \text{ and } A^o(s_{\mathcal{S}}) \subseteq A^o(s_{\mathcal{M}}).$$

A full-control mental model is therefore a deterministic HMI-LTS representing the required information for an operator to interact with a system to the full extent of its possibilities, and without surprises. Full-control mental models are minimal if they have a minimal number of states compared to other full-control mental models of the same system. Also, being *full-control deterministic* is the property of all the systems for which there exists a full-control mental model.

Minimal full-control mental models are important because they represent the minimal model that a perfect operator should learn. Compact training material and user guides should describe a minimal full-control mental model. As already stated the introduction, different algorithms exist to generate such models.

HMI-LTS DECOMPOSITION

While minimal full-control mental models are perfect in terms of control, they are inefficient when training operators as they require to be completely mastered before using a system. We provide a way to split huge models into smaller ones that can be learned independently and reassembled to form larger models.

In this section, we define a new merge operator that combines two HMI-LTSs and we claim that this operator is a natural way to encode the increase of knowledge arising from learning new partial models. We provide a finite decomposition of any HMI-LTS into in a set of basic HMI-LTSs.

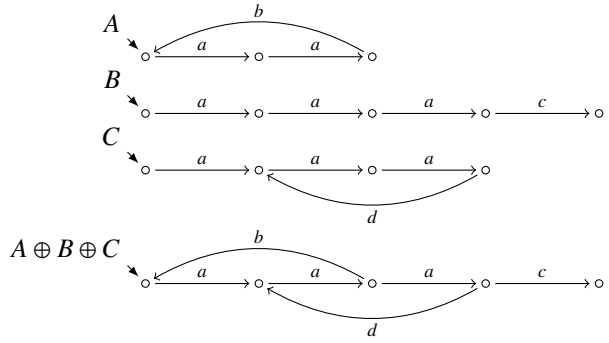

Figure 3. Example of the merge operation on three HMI-LTSs

HMI-LTS merging

Merging two HMI-LTSs produces a third HMI-LTS much like the traditional choice operator, except that common prefixes are merged. This is a kind of lazy choice, as the final behaviour does not commit to behave like the first or the second operand until a decision is required. Notice that the definition does not rely on observations and commands. This definition can therefore be generalized to LTSs.

DEFINITION 5 (MERGE). *The* merge *of two deterministic HMI-LTSs* $A = \langle S_A, \mathcal{L}^c_A, \mathcal{L}^o_A, s_{0A}, \rightarrow_A \rangle$ *and* $B = \langle S_B, \mathcal{L}^c_B, \mathcal{L}^o_B, s_{0B}, \rightarrow_B \rangle$*, denoted* $A \oplus B$*, is an HMI-LTS* $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$ *where* $\mathcal{L}^c = \mathcal{L}^c_A \cup \mathcal{L}^c_B$*,* $\mathcal{L}^o = \mathcal{L}^o_A \cup \mathcal{L}^o_B$ *and* S *is a partition of* $S_A \uplus S_B$ *such that*

1. s_0 *contains at least* $\{s_{0A}, s_{0B}\}$*;*
2. S *is the finest partition of* $S_A \uplus S_B$ *such that for all* $(m, a, m') \in \rightarrow_A$ *and* $(n, a, n') \in \rightarrow_B$ *with* $m, n \in X$ *for some* $X \in S$*, there exists* $Y \in S$ *such that* $\{m', n'\} \subseteq Y$*; and*
3. $\rightarrow$ *is the set of transitions* (X, a, Y) *for which there exists* $x \in X$ *and* $y \in Y$ *such that* $(x, a, y) \in \rightarrow_A$ *or* $(x, a, y) \in \rightarrow_B$*.*

In this definition, S is always well defined. It can be computed by starting with a complete partition where each state is a different element and merging all the states that do not respect the required criterion. This process stops when the criterion is enforced and this happens within a finite number of steps as it must end when the partition contains only one element with all the states in it. The merge of two deterministic HMI-LTS is unique, but this is not necessarily the case in general.

An example of the action of the merge operator is given in figure 3. This example uses the fact that the merge operator is associative. The operator is also commutative. While commutativity can be assumed from the symmetry of the definition, commutativity is more complex and the demonstration is left to the reader.

We can show that the result of merging two deterministic HMI-LTSs is deterministic. Indeed, as the two operands of

the merge are deterministic, they cannot contain τ transitions and so their merge is free of τ transitions too. Let's assume that the result contains a state X such that there exists two transitions with the same label a leading to different states Y and Y'. This means that there exists $(m, a, m') \in \rightarrow_A$ and $(n, a, n') \in \rightarrow_B$ such that $m, n' \in X$, $m' \in Y$ and $n' \in Y'$ which violates the property on S. Namely, m' and n' must belong to the same state Y. The resulting HMI-LTS can contain no τ transitions and no fork where a transition with a same label leads to two different states. This is sufficient to prove that it is deterministic.

The HMI-LTS $A \oplus B$ can switch his behaviour from A to B provided A can reach a state that was merged with a state of B. This conversely holds from B to A. If the HMI-LTS can switch from A to B and from B to A, then it can alternate its behaviour arbitrarily often. We can see that this operator is different from the traditional choice operator because it is more than the union of the traces. It allows to build complex behaviours from two simple models. In figure 3, we can see that the trace $a, a, a, d, a, b, a, a, a, c$ was not possible on the different models but is valid on their merge.

This operator is useful because the set of traces of a merge is always larger or equal to the union of the traces of the merged transitions systems. This means that the possible behaviours of a merge can be richer than the union of the behaviours of its operands. This is needed to ensure that the decomposition of a big system is a small set of small systems. By comparison, the behaviours of a choice are exactly the union of the behaviours of its operands. For synchronous parallel composition, the resulting behaviours are the intersection of the behaviours of the two operands if we synchronise on the union of the alphabets.

Furthermore, the merge operator enforces the interpretation of HMI-LTSs as scenarios. When a scenario loops or terminates, the system is assumed to have returned in a state equivalent to the initial one. In particular, the scenario is assumed to be repeatable infinitely often unless explicitly stated. When an HMI-LTS loops to a given state, it should that the system has returned to a state that is completely equivalent to the initial one for controllability purposes.

The merge operator is therefore great for decomposing systems and is a natural way to encode how mental models grow when learning new ones.

Basic HMI-LTSs

In this section, we explore the decomposition induced by the merge operator on the HMI-LTSs.

The merge operator naturally defines a partial order relation on the HMI-LTSs. The merge order is such that $A \leq_\oplus B$ if and only if $A \oplus B = B$. The strict partial order relation also requires A to be different from B. We can intuitively see that it well defined because merging HMI-LTSs can only increases the set of described behaviours and the merge order captures this.

Furthermore, due to the definition of the merge order, the set of deterministic HMI-LTSs is lattice-structured. Indeed, any

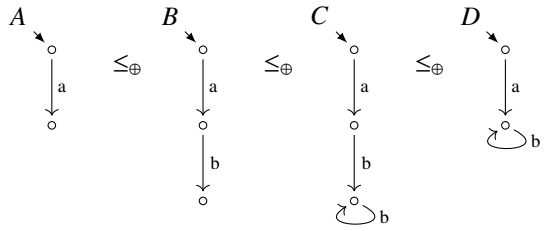

Figure 4. Illustration of the order relation on basic HMI-LTSs. For example, we have $C \leq_\oplus D$ because $C \oplus D = D$.

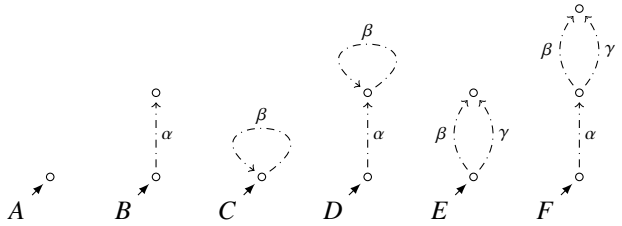

Figure 5. Different shapes of basic HMI-LTSs. They can be A) empty, B) sequences, C) loops, D) lassos and E,F) tulips with and without stem. Dotted lines represent any oriented sequence of states and transitions. All these shapes are degenerated tulips where action sequences α, β and γ can be empty

two HMI-LTSs A and B are (upper) bounded by $A \oplus B$. Therefore, there exists minimal elements called atoms. These are the HMI-LTSs with only one transition. If the lattice was atomistic, we would be able to generate generate all the deterministic HMI-LTSs by merging some of its atoms. This is not the case as we can see in figure 4. There is no way to obtain the graph B by merging HMI-LTSs with only one transistion. However, there exists a larger family of HMI-LTSs that can generate all the deterministic HMI-LTSs, we call them *basic* HMI-LTSs.

DEFINITION 6 (BASIC HMI-LTS). *A deterministic HMI-LTS A is* basic *if it cannot be decomposed into two strictly smaller HMI-LTSs. That is, for all HMI-LTS X, Y such that $X \oplus Y = A$, either $X = A$ or $Y = A$.*

It turns out that such basic HMI-LTS take the form of be single loops, single sequences, lassos or tulips. Loops and sequences can be seen as degenerated lassos with no stem or no loop. The fully degenerated lasso is the HMI-LTS with no transitions at all. Finally, a tulip is a branching HMI-LTS where the two branches reunite in the last state. Like lassos, they may have no stem. All these shapes are drawn in figure 5.

Any finite deterministic HMI-LTS can be decomposed into a finite set of basic HMI-LTS. This arises from the fact that any HMI-LTS is the merge of a basic HMI-LTS and another HMI-LTS strictly smaller than the previous one. Were it not the case, that HMI-LTS would be basic itself. By induction on the remaining HMI-LTS, we show that it eventually reduces to the empty HMI-LTS after a finite number of basic HMI-LTS removal. All the removed basic elements form a set that we call the decomposition of the HMI-LTS.

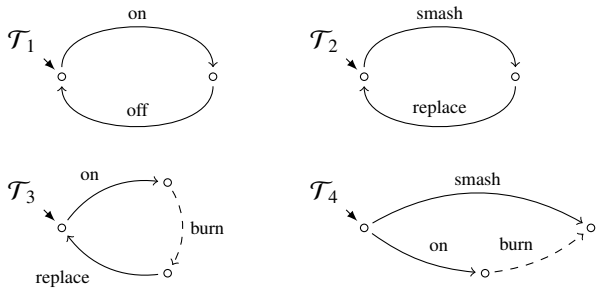

Figure 6. All the basic HMI-LTSs of the Lamp mental model defined in figure 1. $\mathcal{T}_1$ is the only mental model that does not control the Lamp system of figure 1. $\mathcal{T}_{1,2,3}$ are loops and $\mathcal{T}_4$ is a tulip. Amongst the three loops, $\mathcal{T}_1$ represents the fact that the lamp can be switched on and off forever, $\mathcal{T}_2$ that it can be smashed and replaced forever and $\mathcal{T}_3$ that a lamp can be turned on and replaced when it burns to turn it on again. Being a tulip, $\mathcal{T}_4$ has a different meaning. It expresses the fact that smashing a lamp is equivalent to turn it on and observe it burn.

A decomposition is non-redundant if it does not contain two elements such that one is strictly smaller than some other with respect to the merge order defined above. The decomposition algorithm just sketched always produces a non-redundant decomposition because each basic HMI-LTS contains actions that were not part of the previously removed basic elements, and that are removed with it. For example, the decomposition of the Lamp system into $\{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_4\}$ as shown in figure 6 is non redundant.

A decomposition is minimal if no other decomposition of the same HMI-LTS contains less basic elements. The size of a minimal decomposition is called the *complexity* of an HMI-LTS. Minimal decompositions of the Lamp system contain exactly three elements so its complexity is 3. With the basic HMI-LTSs defined in figure 6, we see that $\mathcal{T}_4$ states the equivalence of the "smash" and "on, burn" traces. This implies that $\mathcal{T}_2$ is equivalent to $\mathcal{T}_3$ in a set containing $\mathcal{T}_4$. The two minimal decompositions of the Lamp system are $\{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_4\}$ and $\{\mathcal{T}_1, \mathcal{T}_3, \mathcal{T}_4\}$.

We know how to decompose an HMI-LTS into basic elements, and that decomposition gives us a measure of the complexity of that HMI-LTS.

FULL-CONTROL MENTAL MODEL DECOMPOSITION

In this section, we show that it is possible to build a set of tasks that each control a given model and can be combined into a full-control mental model.

The main idea is to decompose a full-control mental model of the system into basic subgraphs. It appears that basic subgraphs can be completed to form tasks that can control the system. This means that the completed basic subgraphs of a full-control mental model of a system form a set of independent compatible mental models that can be merged to reproduce the behaviour of the full-control mental model.

Basic subgraphs

A subgraph of a graph $\mathcal{G}$ is a graph that contains some of the edges of $\mathcal{G}$. This notion can be extended to HMI-LTSs.

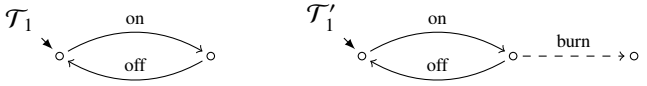

Figure 7. The observation completion of $\mathcal{T}_1$ with respect to the only minimal full-control mental model of Lamp, which is Lamp itself, is $\mathcal{T}'_1$. It contains all the observation transitions from Lamp reachable from $\mathcal{T}_1$. The other basic models $\mathcal{T}_{i,i\neq 1}$ need not be completed. The interpretation of $\mathcal{T}'_1$ is that the lamp can be switched on and off forever as long as it does not burn. When the burning occurs, either the objective is reached or the user is stuck. To unblock the situation, the user can for example read the manual to increase its knowledge of the system or ask a more experienced user.

DEFINITION 7 (SUBGRAPH). *Given two HMI-LTS $\mathcal{T} = \langle S_\mathcal{T}, \mathcal{L}^c, \mathcal{L}^o, s_{0\mathcal{T}}, \rightarrow_\mathcal{T}\rangle$ and $\mathcal{M} = \langle S_\mathcal{M}, \mathcal{L}^c, \mathcal{L}^o, s_{0\mathcal{M}}, \rightarrow_\mathcal{M}\rangle$, $\mathcal{T}$ is a subgraph of $\mathcal{M}$, denoted $\mathcal{T} \subseteq \mathcal{M}$, if $S_\mathcal{T} \subseteq S_\mathcal{M}$, $s_{0\mathcal{T}} = s_{0\mathcal{M}}$ and $\rightarrow_\mathcal{T} \subseteq \rightarrow_\mathcal{M}$*

Given two subgraphs $\mathcal{T}$ and $\mathcal{T}'$ of an HMI-LTS $\mathcal{M}$, we have the nice property that their merge $\mathcal{T} \oplus \mathcal{T}'$ is a subgraph of $\mathcal{M}$ up to a relabelling of the states. That is, the merge of two subgraphs of $\mathcal{M}$ is isomorphic to some subgraph of $\mathcal{M}$. This can be seen from the fact that $\oplus$ merges states that can be reached with the same traces, and that states must correspond to the same state of $\mathcal{M}$ as $\mathcal{M}$ is deterministic.

Therefore, any HMI-LTS can be decomposed into a non-redundant finite set of its basic subgraphs.

Tasks

Starting from a full-control mental model $\mathcal{M}$ of a system $\mathcal{S}$, we can decompose it into a set of basic HMI-LTSs. However, these basic HMI-LTSs do not necessarily control $\mathcal{S}$. To achieve this property, they need to be completed with respect to observations. This holds because of the symmetric nature of the control property. A mental model that controls a system must accept all the observations of that system, but is allowed to ignore commands.

If we call $\rightarrow^o_S$ the transition relation of $\mathcal{S}$ restricted to observations, then any basic subgraph of a full-control mental model can be completed with $\rightarrow^o_S$ in order to control $\mathcal{S}$. Of course, only the connected component reachable from the initial state should be kept after the completion. Figure 7 shows the completion of the basic HMI-LTS $\mathcal{T}_1$ from figure 6.

DEFINITION 8 (OBSERVATION COMPLETION). *Given an HMI-LTS $\mathcal{M} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow^c \cup \rightarrow^o\rangle$ and one subgraph $\mathcal{T} = \langle S_\mathcal{T}, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow_\mathcal{T}\rangle$ of $\mathcal{M}$, the* observation completion *of $\mathcal{T}$ is an HMI-LTS $\mathcal{T}'$ such that $\mathcal{T}'$ is the connected component of $\langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow_\mathcal{T} \cup \rightarrow^o\rangle$ reachable from s_0.*

The observation completion of any subgraph of a full-control mental model $\mathcal{M}$ controls the intended system. Indeed, such a completed subgraph cannot prevent observations from occurring as the full-control mental model does not, and the completed graph has all the observations from the system. In particular, the observation completion of basic subgraphs of full-control mental models of a system $\mathcal{S}$ control that system $\mathcal{S}$. These elements also have the nice property of merging

into completed subgraphs of $\mathcal{M}$ that themselves have control over $\mathcal{S}$.

DEFINITION 9 (BASIC TASK).
Given a full-control mental models $\mathcal{M} = \langle S, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow \rangle$ where $\rightarrow = \rightarrow^c_{\mathcal{M}} \cup \rightarrow^o_{\mathcal{M}}$, a basic task is a mental model $\mathcal{T} = \langle S_{\mathcal{T}}, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow_T \rangle$ such that $\rightarrow_T = \rightarrow^o_{\mathcal{M}} \cup \rightarrow_b$ and $\langle S_{\mathcal{T}}, \mathcal{L}^c, \mathcal{L}^o, s_0, \rightarrow_b \rangle$ is a basic subgraph of $\mathcal{M}$.

With this definition, we can state that any full-control deterministic system is fully controlled by the merge of a set of basic tasks. As an example, $\mathcal{T}'_1$, $\mathcal{T}_2$ and $\mathcal{T}4$ form such a set from figures 6 and 7 form such a set.

THEOREM 1 (TASK DECOMPOSITION). *Any finite fc-deterministic HMI-LTS $\mathcal{S}$ can be decomposed into a finite set $T = \{\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n\}$ of basic tasks such that*

- *each $\mathcal{T}_i$ controls $\mathcal{S}$;*
- *for each subset $I \subset \{1, 2, \ldots n\}$ of indices, the partial merge $\bigoplus_{i \in I} \mathcal{T}_i$ of elements of T controls $\mathcal{S}$; and*
- *the complete merge $\bigoplus_{i=1}^{n} \mathcal{T}_i$ has full-control over $\mathcal{S}$.*

PROOF. By definition, any fc-deterministic HMI-LTS $\mathcal{S}$ has at least one minimal full-control mental model $\mathcal{M}$. We have shown that such a full-control mental model can be decomposed into a finite set of basic tasks which are completed basic subgraphs. Because these elements are completed subgraphs they have control over $\mathcal{S}$ and any partial merge of these elements have too. As the elements are the completion of the decomposition of $\mathcal{M}$ into basic HMI-LTS, their full merge will be exactly $\mathcal{M}$, and therefore fully controls $\mathcal{S}$. This proves that there exists a decomposition of $\mathcal{S}$ meeting the required properties. □

Task-complexity

The decomposition of an fc-deterministic system into a set of tasks is far from unique. Indeed, there exists an infinity of full-control mental models for a given system, and for each full-control mental model there may exist multiple decompositions into basic tasks.

Nevertheless, we define the task-complexity of a system as the size of the smallest set of tasks that can be merged into a full-control mental model of that system. This metric measures the number of small tasks that an operator needs to learn before being able to control all the features of the system.

This metric is different from both the number of states and the number of transitions which are the most common measures of transition systems.

CONCLUSION

We have defined the merge operation that represents how a human augments its mental model by leaning new mental models. We have shown that this operation is more natural than the parallel synchronisation operation and more powerful than the classical choice operation.

We have shown how tasks can be split into simple operations. In particular, we have shown that the full-control mental model of a system is itself a composition of basic tasks. With this decomposition, we have defined a measure of the complexity of HMI-LTSs based on tasks.

We know how to generate decompositions of a system into tasks, and we know that there exists a minimal decomposition, but we do not yet know how to generate minimal decompositions. In the near future, we expect to work on an algorithm capable of computing one.

This works can be used to validate system design by detecting irreducible large tasks. But, more importantly, this works lays the ground for automated generation of user manuals and assisted generation of other training material. With small well-defined tasks, it is possible to decompose manuals into self-contained chapters.

REFERENCES

1. Sébastien Combéfis. 2013. *A Formal Framework for the Analysis of Human-Machine Interactions*. Vol. 459. Presses universitaires de Louvain.
2. Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, and Michael Feary. 2011a. A formal framework for design and analysis of human-machine interaction. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*. IEEE, 1801–1808.
3. Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, and Michael Feary. 2011b. Learning system abstractions for human operators. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*. ACM, 3–10.
4. Sébastien Combéfis and Charles Pecheur. 2009. A bisimulation-based approach to the analysis of human-computer interaction. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 101–110.
5. Philippe A Palanque, Rémi Bastide, and Valérie Sengès. 1995. Validating interactive system design through the verification of formal task and system models.. In *EHCI*. 189–212.
6. Peter J Ramadge and W Murray Wonham. 1987. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization* 25, 1 (1987), 206–230.
7. John Rushby. 2002. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering & System Safety* 75, 2 (2002), 167–177.
8. Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing*, Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.). Lecture Notes in Computer Science, Vol. 4949. Springer Berlin Heidelberg, 1–38.

Towards formal modeling of App-Ensembles

Johannes Pfeffer
Technische Universität Dresden
Dresden, Germany
johannes.pfeffer@tu-dresden.de

Leon Urbas
Technische Universität Dresden
Dresden, Germany
leon.urbas@tu-dresden.de

ABSTRACT
App-Ensembles are interactive systems comprised of several self-contained apps that are connected in a purposeful manner via navigation links and input/output channels. This paper describes the current state of development of a formal modeling language for App-Ensembles. The development is part of the Application Orchestration Framework Language (AOF-L) that includes other modeling languages based on the same vocabulary, e.g. for semantic description of apps. In the presented research, the focus lies on the graphical representation of App-Ensembles. The graphical modeling elements used in the developed language are taken exclusively from BPMN 2.0. It is shown that App-Ensembles can be easily integrated into classical business process models modeled in BPMN 2.0. The utility of App-Ensembles and the practicality of the modeling approach are demonstrated via a use case example covering maintenance tasks in a nuclear power plant. Finally, an approach to apply a formal BPMN modeling approach to App-Ensembles is presented.

Author Keywords
App-Ensembles, App-Orchestration, BPMN, Formal Methods, Formal Modeling, Interactive Systems, Linked Data, Mobile Information Systems, Semantic Web, Workflows.

ACM Classification Keywords
D.2.1. Requirements/Specifications: Languages; D.2.2. Design Tools and Techniques: User interfaces; H.1.2 User/Machine Systems; I.5.5. Model Development: Modeling methodologies; D.2.9. Management: Software process models

INTRODUCTION
Originally, the term *app* was just an abbreviation for the word application. Today, in times of ubiquitous mobile devices the notion of *app* usually doesn't refer to a general software application, as e.g. an operating system or a web server. Apps are specialized programs, mostly used on mobile devices that assist users in completing tasks that have a clearly defined scope. Early apps were used by delivery services in the late nineties on Windows CE devices to capture signatures for parcel reception confirmation. At that time, these monolithic apps served their purpose well. Even today, delivery people can still be seen working with these systems. However, as workflows become more dynamic and change more regularly in structure, composition, and goals, rigid user interfaces for mobile information systems are tedious to keep up to date and may pose a burden to implementing agile work processes.

This paper presents research towards establishing a formal model for interactive systems comprised of self-contained apps that are connected in a purposeful manner via navigation links and input/output channels (*App-Ensembles*).

The paper is organized as follows. In the following section, the background for the presented work is briefly described. Then the AOF-Language is introduced and its graphical notation is explained. Next, the textual notation of the language using Semantic Web technologies is shown. Then the graphical language is applied to a use case example for a maintenance task in a nuclear power plant. Afterwards an approach for formal modeling of App-Ensembles is described. The presented work is discussed and the paper closes with a conclusion and outlook.

BACKGROUND
Pfeffer et al. [9] describe a concept to orchestrate apps into interactive systems that can support users in performing complex tasks. The approach addresses the challenge to generate flexible, adaptable and usable app-based user interfaces with minimal effort. It takes into account that data sources, workflows, users and the apps themselves may be subject to continuous change. The approach heavily relies on the Semantic Web stack [2]. For easy integration in existing business processes and virtual manufacturing networks, it is based on established concepts from the field of business process modeling.

The *App-Orchestration* process [12] consists of three steps: *Select*, *Adapt*, and *Manage* (as shown in Figure 1). The process relies on a set of semantically described mobile apps, a well-defined information space, and a business process model that defines necessary tasks and their relations. In the *Select* step, a subset of apps is selected from an app pool that best support the tasks given in the

To cite this article use the following information:

Johannes Pfeffer and Leon Urbas. 2015. Towards formal modeling of App-Ensembles. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 13-18.
http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

actual business process. To increase reusability, in addition to commercial off-the-shelf commercial apps, the app pool may contain generic apps made for orchestration that are not yet adapted to a specific use case or information source. Instead, they are adaptable to various specific tasks. This is done in the *Adapt* step. Adaptation may include the visual appearance, the interaction style, the actual data source, and others. In the *Manage* step, the adapted apps are connected according to a navigation design model which is derived from the business process model. Therefore, only useful navigation paths are presented to the user.

All three steps are performed at design time. As a result, a deployable *App-Ensemble* consisting of a model and installable app artifacts is produced. The App-Ensemble is then deployed to a device (e.g. an Android tablet). At runtime, a workflow execution engine enforces the navigation design (switching from app to app, allowing the user to make decisions at workflow branches) and facilitates information exchange between the apps. The same set of apps can be orchestrated in many ways, depending on the needs of the underlying business process.

The set of tools and specifications for App-Orchestration is collected in the Application Orchestration Framework (AOF). The modeling language to describe apps, App-Ensembles and Orchestration is called AOF-Language (AOF-L).

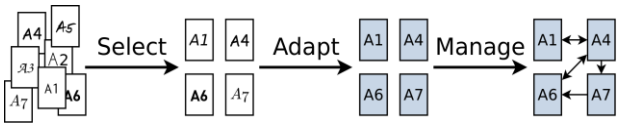

Figure 1. App-Orchestration steps - Select, Adapt, Manage

THE AOF-LANGUAGE

The AOF-Language is currently in development and consists of two parts. The first part defines language elements that allow semantic description of properties of an app. This includes name, textual description, version, and others. Also, the interface of the app is described (entry points, exit point, inputs, outputs) and other information needed to start the app, provide data to it and receive results. This part of the language is not a major subject of this paper. The second part which is the main focus of this paper allows the formal description of an App-Ensemble. The language specification [14] also includes a formal semantic vocabulary (AOF-Vocabulary) specified using RDFS [13] and concepts from OWL 2 [15].

App-Ensembles are described using a graphical notation that is based on BPMN 2.0 [16]. This paper is only concerned with version 2.0 of BPMN. Therefore, from here on the version is omitted for brevity.

The combination of a well-known business process modeling language and Semantic Web technologies makes it possible to easily publish BPM and App-Ensembles in form of Linked Data [2,4]. This enables integration in collaborative Virtual Enterprises as described in [8].

For execution the model is serialized as RDF using concepts defined in the AOF-Vocabulary, from a BPMN-Ontology [10], Friend of a friend (FOAF) [17], Dublin Core (DC) [6], RDF and RDFS [13]. There is a graphical representation which is exclusively based on modeling elements of BPMN. This paper is mainly concerned with the graphical notation because it is well-suited to illustrate an App-Ensemble model for a use case.

Graphical Modeling Elements

AOF-L uses a subset of BPMN modeling elements as shown in Figure 2. BPMN modeling elements other than those shown are not allowed. All BPMN connection rules that apply to the allowed modeling elements are valid (for an overview see [16], p. 40ff). None of the semantics of BPMN are contradicted.

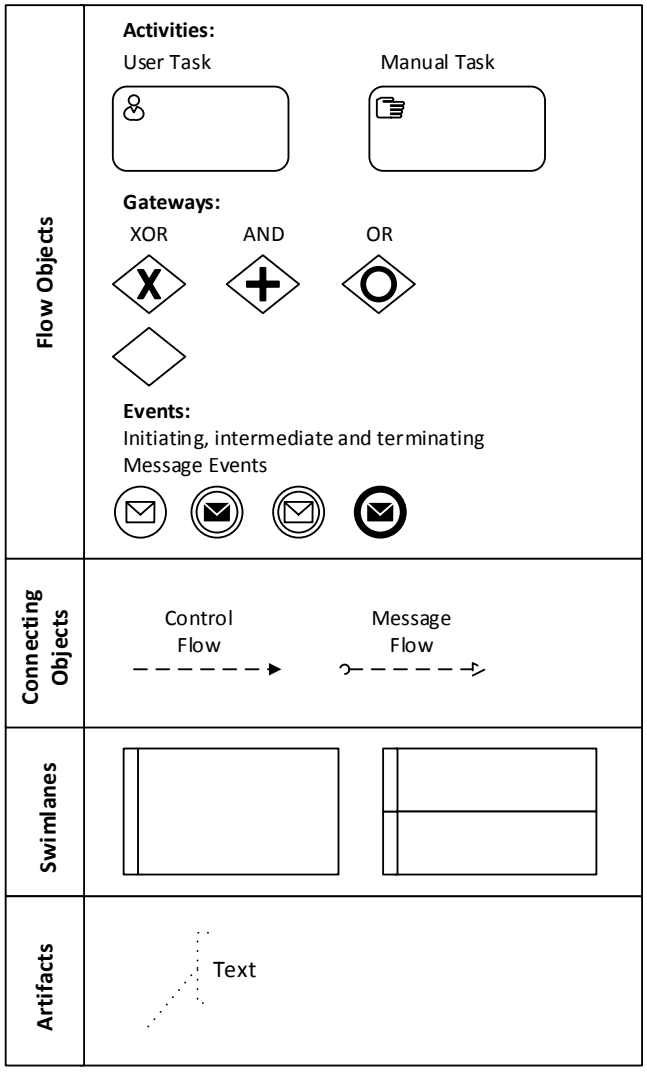

Figure 2: Subset of BPMN modeling elements used in AOF-L

Activities

By definition, a *User Task* is executed by a human being through a user interface. In AOF-L this task type is used to represent an app in an App-Ensemble. The Activity type User Task is extended in compliance with the BPMN specification ([16], p 55) with a reference to a semantic app description. The activity is completed when the app is closed by the user or an interaction with the app is performed that yields output values.

A *Manual Task* is executed by a human without the help of an app or other type of user interface. A typical example in the use case domain would be to open or close a valve. By using a Manual Task in an App-Ensemble tasks can be modeled that users have to perform without assistance of an app. They are treated as tasks and have to be confirmed as completed to the execution engine before the control flow continues past the activity.

Gateways

Gateways are used to split or join a workflow. The flow of control is affected in the same manner as in standard BPMN. However, due to the fact that User Tasks represent apps that run on a physical device and can be used by a user only one at a time, no real parallelism is possible. Thus, when two or more branches with User Tasks are activated at the same time as a result of a split, they are placed in an *App-Dash*. The App-Dash is a list of available apps – comparable to a task list in classical business process execution engines. Where appropriate, the user is asked by the process execution engine to choose between possible next apps to continue the workflow.

The behavior of the gateways in an App-Ensemble is described in the following paragraphs. This is a specialization of the behavior specified by BPMN.

An *XOR-Split* splits an App-Ensemble workflow into multiple branches (see Figure 3). Exactly one of the branches is activated based on user decision. The other branches are not activated. An *XOR-Join* joins multiple workflow branches. The workflow continues as soon as one of the incoming branches is activated. In an App-Ensemble, this occurs when at least one app (User Task) directly preceding the gateway finishes.

The *AND-Split* splits an App-Ensemble workflow into multiple branches that are all activated at the same time. This results in multiple new apps placed in the users App-Dash (see Figure 3). An *AND-Join* allows the workflow to continue when all of the incoming branches are activated, i.e. when all preceding apps have finished.

For an *OR-Split* the App-Ensemble workflow is split based on user decision. The user may decide to activate one or more branches resulting in one or more apps being placed in the App-Dash (see Figure 3). At an *OR-Join* the workflow is continued based on user decision when at least one incoming branch has been activated.

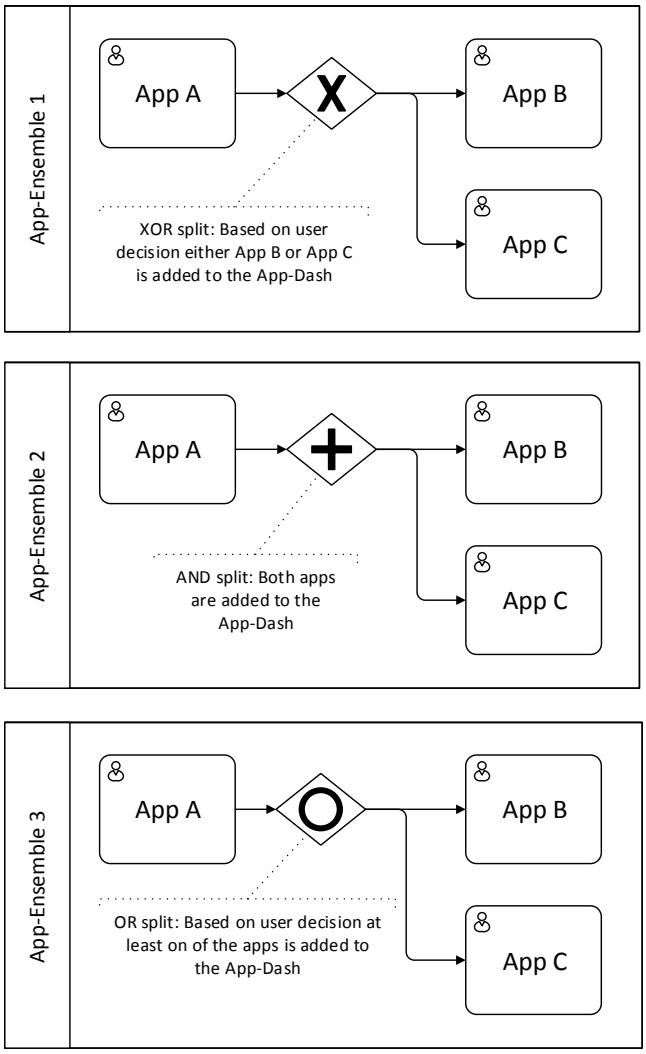

Figure 3: Branching XOR, AND & OR gateways

Events

Data can be received from a superordinate Process using *Throwing Message Events* (intermediate or initiating). Data can be sent to a superordinate process using *Catching Message Events* (intermediate or terminating).

Connecting Objects

The *Control Flow* arrow is used just as in BPMN. *Message Flow* arrows are used to connect an App-Ensemble with superordinate business processes. They are used to synchronize the business process with the App-Ensemble.

Swimlanes

Optionally, App-Ensembles may be placed in their own *Pool*. They must be placed in their own exclusive Pool whenever the BPM describes a collaboration (more than one participant). *Lanes* may be used to organize the modeling elements in the Pool but have, as in BPMN, no relevance to the control flow.

Start and End Events are optional according to the BPMN specification (see [16], p. 237 and p. 246). In AOF-L they

are not allowed, except for initiating and terminating message events.

Textual Notation of the Model

The notation of an App-Ensemble model is facilitated using RDF. Since the focus of this paper is the graphical notation for a specific use case, this shall be illustrated only briefly by the example of a single User Task representing an app. Using the BPMN ontology [10] and the AOF-Vocabulary an instance of a User Task can be described in Turtle RDF notation [1] as follows:

```
@prefix aof: <http://eatld.et.tu-dresden.de/aof/>.
@prefix bpmn2: <http://dkm.fbk.eu/index.php/
                                BPMN2_Ontology#>.
@prefix ae: <http://example#>.

ae:userTask1 a aof:App_1;
  bpmn2:id "App_1";
  bpmn2:name "App 1".
```

In the listing, `ae:userTask_1` is an instance of `aof:App_1` (in Turtle "a" is an abbreviation for `rdf:type` which describes an instance-of relation). `aof:App1` is a resource describing the app semantically. The semantic description describes the interface of the app by defining entry points with input variables of a certain type and exit points with output variables (for an example of a semantic app description see [14]).

All other modeling elements can be written in the same fashion. Naturally, any other RDF serialization such as N3 or RDF-XML can be used in place of Turtle.

USE CASE EXAMPLE

The use case presented here is loosely based on Use Case 1, *Control of a Nuclear Power Plant*, from the FoMHCI workshop call. A nuclear power plant is a special type of process plant with very high security requirements. However, like any other process plant, it consists of machines, devices, physical connections (pipes, wires, etc.) sensors and other equipment. The use case focuses on a fraction of a business process model (BPM) that models maintenance tasks for the feedwater pumps of the nuclear power plant, see Figure 4.

Existing BPM can be adapted for execution with App-Ensembles by adding message events before and after tasks that are completed with the help of apps. In the use case, the BPM is triggered when the maintenance interval for the feedwater pump is reached. The presented model has been simplified for reasons of brevity but the basic workflow is realistic.

At the beginning, the maintenance personnel checks the state of the safety valves and performs a visual inspection of the pump (e.g. for leakage). The latter two tasks are manual and are not supported by any tool controlled by the process. If something notable is observed a report on the findings is created. Finally, the pump diagnosis is performed and another report is written. In a BPM that is not supported by an App-Ensemble, the process looks just the same, with the exception of the additional send and receive events. Thus, existing BPM can be easily extended for use with App-Ensembles by adding a participant (in BPMN terminology this refers to separate process) containing the App-Ensemble model to the collaboration.

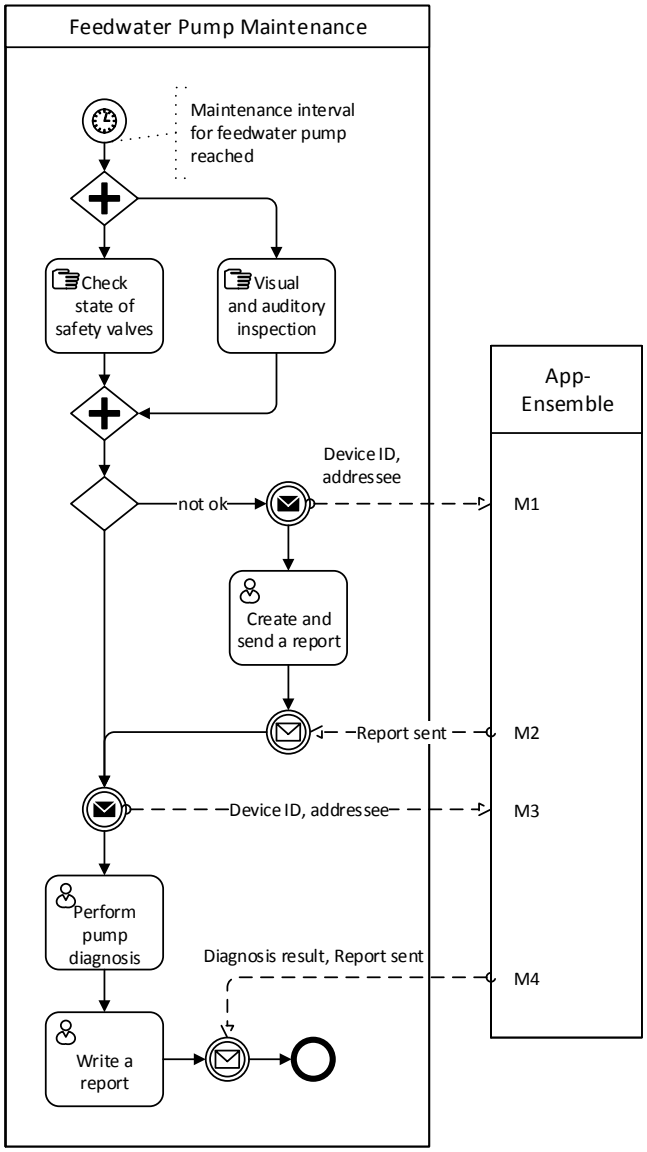

Figure 4: BPM perspective on the maintenance workflow

In Figure 5, the process is modeled from the perspective of the App-Ensemble. The maintenance personnel uses two apps for diagnosing the pump. The control flow starts with a token being generated either on the receiving start message event or the receiving intermediate message event. In the first case, the control flow splits into two parallel branches and the token is cloned. Now the user must serialize this parallelism by choosing the order in which the apps shall be presented by selecting an app from the App-Dash. Only when both apps have been used (i.e. the activity has been completed) and expected data has been yielded,

the control flow continues and a report can be created. There is a second entry point to the App-Ensemble that is triggered when a report has to be written after preliminary inspection of the pump. This shows that parts of App-Ensembles can be flexibly reused and may be instantiated multiple times. After the result data set (which may consist of the diagnosis result and/or the report) is returned to the superordinate process, the token is consumed.

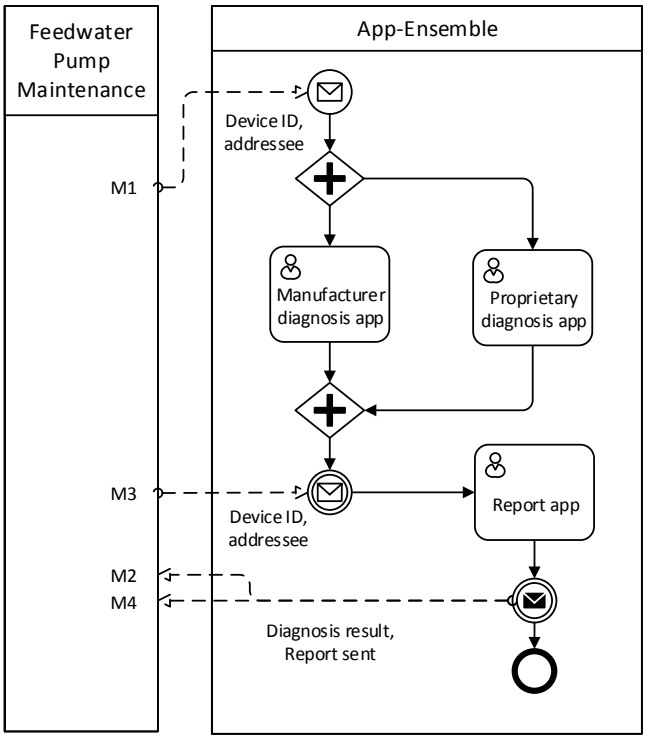

Figure 5: App-Ensemble perspective on the maintenance workflow

FORMAL MODELING

On the one hand, the AOF-Language is RDF-based and represents an RDF graph. On the other hand, it makes heavy use of concepts from BPMN. Thus, it is self-evident to evaluate formalization via methodologies that have been used to formalize business process models. In the following section this is investigated in order to discuss the approach during the workshop.

BPMN-based model

Since App-Ensembles can be expressed via a subset of BPMN modeling elements it is possible to formally model an App-Ensemble process along the lines described in [3]:

An App-Ensemble process is a tuple $\mathcal{P} =$

$(\mathcal{O}, \mathcal{A}, \mathcal{E}, \mathcal{G}, \{t^U\}, \{t^M\}, \{e^S\}, \{e^{I_R}\}, \{e^{I_S}\}, \{e^E\}, \mathcal{G}^S, \mathcal{G}^J, G^{O_S},$

$\mathcal{G}^{O_J}, \mathcal{G}^X, \mathcal{G}^M, \mathcal{F})$ *where:*

$\mathcal{O}$ *is a set of objects which can be partitioned into disjoint sets of activities* $\mathcal{A}$*, events* $\mathcal{E}$*, and gateways* $\mathcal{G}$*,*

$\mathcal{A}$ *can be partitioned into disjoint sets of user tasks* $\{t^U\}$ *and manual tasks* $\{t^M\}$*,*

$\mathcal{E}$ *can be partitioned into disjoint sets of start message events* $\{e^S\}$*, receiving intermediate message events* $\{e^{I_R}\}$*, sending intermediate message events* $\{e^{I_S}\}$*, and end message events* $\{e^E\}$*,*

$\mathcal{G}$ *can be partitioned into disjoint sets of splitting AND gateways* $\mathcal{G}^S$*, joining AND gateways* $\mathcal{G}^J$*, splitting OR gateways* G^{O_S}*, joining OR gateways* $\mathcal{G}^{O_J}$*, splitting XOR gateways* $\mathcal{G}^X$*, and joining XOR gateways* $\mathcal{G}^M$*.*

$\mathcal{F} \subseteq \mathcal{O} \times \mathcal{O}$ *is the control flow relation, i.e. a set of sequence flows connecting objects.*

Within this definition, an App-Ensemble is a directed graph with nodes $\mathcal{O}$ and arcs $\mathcal{F}$. For any node $x \in \mathcal{O}$, input nodes of x are given by $in(x) = \{\in \mathcal{O} \mid y\mathcal{F}x\}$ and output nodes of x are given by $out(x) = \{\in \mathcal{O} \mid x\mathcal{F}y\}$.

Based on this formal syntax, an App-Ensemble Model can be formulated and requirements for it being well-formed can be defined. As shown in [3] the model can now be mapped to Petri nets.

DISCUSSION

It should be noted that due to ambiguities in the BPMN specification in certain cases the mapping may be problematic. For instance when multiple start events occur it is not clearly defined whether a new process instance should be created or a new token should be created in the currently running instance. However, aside from these ambiguities, transformation to Petri nets opens a rich and well-established set of tool for analysis and verification of App-Ensembles.

The App-Ensemble model relies on the formal BPMN-ontology [10]. Consequently, it inherits many of its strengths as well as its limitations. Strengths include the ability to run reasoning services on the model (e.g. for consistency checking or inferring implicit types), the ability to query the model using languages such as SPARQL, and the ability to perform a formal verification regarding contradictions and BPMN compliance. Due to OWL 2 limitations, some conditions and default values described in the BPMN specification are not encoded in the ontology (see [10], p. 5f).

App-Ensembles are interactive systems of user interfaces. The granularity of the App-Ensemble model ends at the level of the app – a self-contained user interface that is tailored for tending to a well contained need. It should be investigated if the notion of *app* can be seen as a specialization of the term *user interface* as defined in [11]. If so, instances of user interfaces in App-Ensembles could be modeled using the FILL and VFILL language. While AOF-L is restricted to modeling input and output of an app that is provided by a business process, FILL can model input and output that is generated or consumed by the user

(e.g. changing diagnosis parameters and viewing the results). By linking a FILL model of an app user interface to a semantic app-description a more complete model of the whole interactive system could be created.

CONCLUSION AND OUTLOOK

This research has presented advancements towards a language for modeling App-Ensembles. Its application in a use case has been shown to substantiate the ability of the language to model interactive systems of user interfaces. The ontological formalization of the AOF-L is not yet complete.

Currently, the authors are working on extending the AOF-Vocabulary towards a lightweight ontology and joining it with the BPMN-ontology to create a unified formal ontology of the complete AOF-Language.

Since AOF-L is an RDF-based language, RDF-based approaches to formalization should also be considered. In [5] bipartite Graphs are used as an intermediate model for RDF. The authors of [7] model RDF as a directed hypergraph which is a very concise and expressive model for RDF graphs. These approaches will be investigated in coming research.

ACKNOWLEDGEMENTS

The research leading to the presented work was partially funded by the German Ministry of Education and Research (BMBF Grant No. 01IS14006A).

REFERENCES

1. Beckett, D. and Berners-Lee, T. Turtle-terse RDF triple language. *W3C Team Submission 14*, (2008).
2. Bizer, C., Heath, T., and Berners-Lee, T. Linked data-the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS) 5*, 3 (2009), 1–22.
3. Dijkman, R.M., Dumas, M., and Ouyang, C. Formal semantics and analysis of BPMN process models using Petri nets. *Queensland University of Technology, Tech. Rep*, (2007).
4. Graube, M., Ziegler, J., Urbas, L., and Hladik, J. Linked data as enabler for mobile applications for complex tasks in industrial settings. *Proc. 18th IEEE Conf. Emerging Technologies Factory Automation (ETFA)*, (2013), 1–8.
5. Hayes, J. and Gutierrez, C. Bipartite Graphs as Intermediate Model for RDF. .
6. ISO-15836. Information and documentation - The Dublin Core metadata element set. 2009. https://www.iso.org/obp/ui/#iso:std:iso:15836:ed-2:v1:en.
7. Morales, A.A.M. and Serodio, M.E.V. A directed hypergraph model for RDF. *Proc. of Knowledge Web PhD Symposium*, (2007).
8. Münch, T., Hladik, J., Salmen, A., et al. Collaboration and Interoperability within a Virtual Enterprise Applied in a Mobile Maintenance Scenario. *Revolutionizing Enterprise Interoperability through Scientific Foundations*, (2014), 137–165.
9. Pfeffer, J., Graube, M., Ziegler, J., and Urbas, L. Networking apps for complex industrial tasks - Orchestrating apps efficiently. *atp edition 55(3)*, (2013), 34–41.
10. Rospocher, M., Ghidini, C., and Serafini, L. An ontology for the Business Process Modelling Notation. *Formal Ontology in Information Systems - Proceedings of the Eighth International Conference, FOIS2014, September, 22-25, 2014, Rio de Janeiro, Brazil*, IOS Press (2014), 133–146.
11. Weyers, B. *Reconfiguration of User Interface Models for Monitoring and Control of Human-Computer Systems.* Dr. Hut Verlag, Berlin, 2012.
12. Ziegler, J., Pfeffer, J., Graube, M., and Urbas, L. Beyond App-Chaining: Mobile App Orchestration for Efficient Model Driven Software Generation. *Proceedings of the 17th international IEEE Conference on Emerging Technologies & Factory Automation*, (2012).
13. Resource Description Framework (RDF): Concepts and Abstract Syntax. 2014. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.
14. AOF Language specification version 002. http://eatld.et.tu-dresden.de/aof/spec/.
15. OWL 2 Web Ontology Language Document Overview (Second Edition). http://www.w3.org/TR/owl2-overview/.
16. OMG. Business Process Modeling Notation, v.2.0 - Specification. http://www.omg.org/spec/BPMN/2.0/.
17. FOAF Vocabulary Specification. http://xmlns.com/foaf/spec/.

Beyond Formal Methods for Critical Interactive Systems: Dealing with Faults at Runtime

Camille Fayollas[2,3], Célia Martinie[2], Philippe Palanque[2], Yannick Deleris[1]

[1] AIRBUS Operations,
316 Route de Bayonne,
31060, Toulouse, France
yannick.deleris@airbus.com

[2]ICS-IRIT, University of Toulouse,
118 Route de Narbonne,
F-31062, Toulouse, France
{Name}@ irit.fr

[3]LAAS-CNRS,
7 avenue du colonel Roche,
F-31077 Toulouse, France
cfayolla@laas.fr

ABSTRACT

Formal methods provide support for validation and verification of interactive systems by means of complete and unambiguous description of the envisioned system. They can also be used (for instance in the requirements/needs identification phase) to define precisely what the system should do and how it should meet user needs. If the entire development process in supported by formal methods (for instance as required by DO 178C [7] and its supplement 333 [8]) then classical formal method engineers would argue that the resulting software is defect free. However, events that are beyond the envelope of the specification may occur and trigger unexpected behaviors from the formally specified system resulting in failures. Sources of such failures can be permanent or transient hardware failures, due to (when such systems are deployed in the high atmosphere e.g. aircrafts or spacecrafts) natural faults triggered by alpha-particles from radioactive contaminants in the chips or neutron from cosmic radiation. This position paper proposes a complementary view to formal approaches first by presenting an overview of causes of unexpected events on the system side as well as on the human side and then by discussing approaches that could provide support for taking into account system faults and human errors at design time.

Author Keywords

Formal methods; interactive systems; human reliability; fault-tolerant systems.

ACM Classification Keywords

D.2.2 [Software] Design Tools and Techniques – Computer aided software engineering (CASE).

INTRODUCTION

The overall dependability of an interactive system is the one of its weakest component and there are many components in such systems ranging from the operator processing information and physically exploiting the hardware (input and output devices), interaction techniques, to the interactive application and possibly the underlying non interactive system being controlled.

Building reliable interactive systems is a cumbersome task due to their very specific nature. The behavior of these reactive systems is event-driven. As these events are triggered by human operators, these systems have to react to unexpected events. On the output side, information (such as the current state of the system) has to be presented to the operator in such a way that it can be perceived and interpreted correctly. Lastly, interactive systems require addressing together hardware and software aspects (e.g. input and output devices together with their device drivers).

In the dependable computing domain, empirical studies have demonstrated (e.g. [20]) that software failures may occur even though the development of the system has been extremely rigorous. One of the many sources of such failures is called natural faults [1] triggered by alpha-particles from radioactive contaminants in the chips or neutron from cosmic radiation. A higher probability of occurrence of faults [31] concerns systems deployed in the high atmosphere (e.g. aircrafts) or in space (e.g. manned spacecraft [13]). Such natural faults demonstrate the need to go beyond classical fault avoidance at development time (usually brought by formal description techniques and properties verification) and to identify all the threats that can impair interactive systems.

WHY FORMAL METHODS AND ZERO DEFECT APPROACHES ARE NOT ENOUGH

To be able to ensure that the system will behave properly whatever happens, a system designer has to consider all the issues that can impair the functioning of that system. In the perspective of identifying all of them, in the domain of dependable computing, Avizienis et al [1] have defined a typology of faults. This typology leads to the identification of 31 elementary classes of faults. Figure 1 presents a simplified view of this typology and makes explicit the two main categories of faults (top level of the figure): i) the ones occuring at development time (including bad designs, programming errors, …) and ii) the one occuring at opera-

To cite this article use the following information:

Camille Fayollas, Célia Martinie, Philippe Palanque, and Yannick Deleris. 2015. Beyond Formal Methods for Critical Interactive Systems: Dealing with Faults at Runtime. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 19-23.
http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

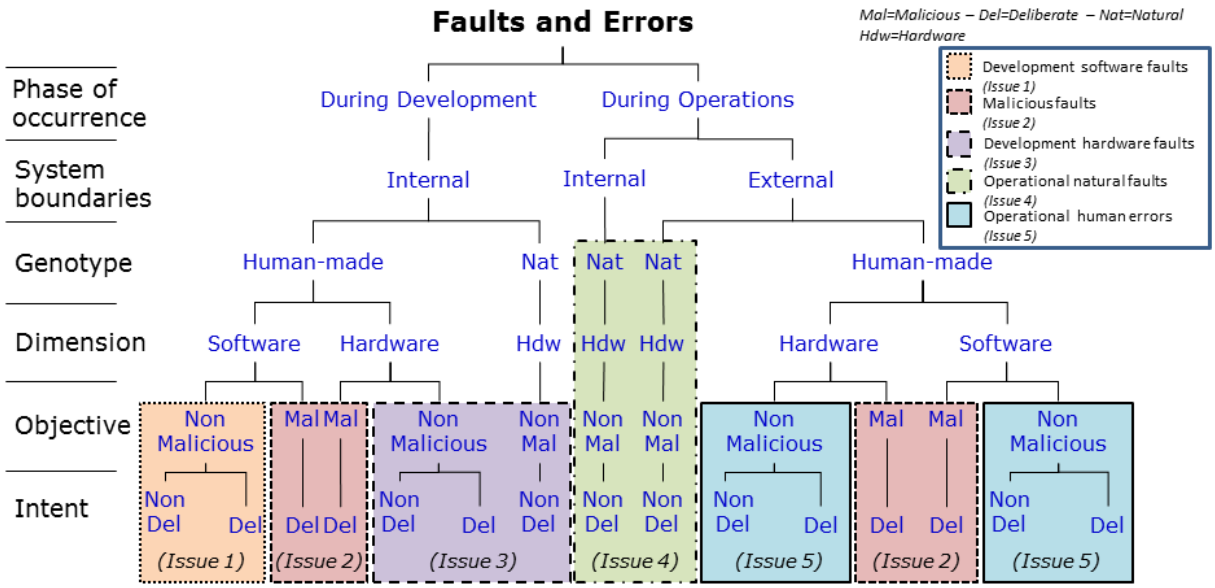

Figure 1. Typology of faults in computing systems (adapted from [1]) and associated issues for the resilience of these systems

tion times (right-hand side of the figure including user error such as slips, lapses and mistakes as defined in [25]).

We propose to organize the leaves of the typology in five different groups as each of them brings a special problem (issue) to be addressed:

- *Development software faults (issue 1)*: software faults introduced by a human during the system development.
- *Malicious faults (issue 2)*: faults introduced by human with the deliberate objective of damaging the system (e.g. causing service denial or crash of the system).
- *Development hardware faults (issue 3)*: natural (e.g. caused by a natural phenomenon without human involvement) and human-made faults affecting the hardware during its development.
- *Operational natural faults (issue 4)*: faults caused by a natural phenomenon without human participation, affecting the hardware and occurring during the service of the system. As they affect hardware, they are likely to damage software as well.
- *Operational human-errors (issue 5)*: faults resulting from human action during the use of the system. These faults are particularly of interest for interactive system and the next subsection describe them in detail.

We consider that malicious faults are beyond the scope of this position paper and will thus not be further discussed. However, it might be interesting within the workshop to address this aspect that is more and more relevant with the open, collaborative interactive systems.

Considering system faults

In the domain of fault-tolerant systems, empirical studies have demonstrated (e.g. [20]) that software crashes may occur even though the development of the system has been extremely rigorous. One of the many sources of such crashes is called natural faults [1] triggered by alpha-particles from radioactive contaminants in the chips or neutron from cosmic radiation. A higher probability of occurrence of faults [31] concerns systems deployed in the high atmosphere (e.g. aircrafts) or in space (e.g. manned spacecraft [13]). Furthermore the evolution of modern IC components may lead in the next future to a higher probability of physical faults in operation. Although the recommendation for avionics systems is 100 FITs over 25 years lifetime, the current Deep Sub-Micron (DSP) technology may lead to a failure rate up to 1000 FITs, only during 5 years operational life time [28]. This is major worry in the avionics industry since this tendency has two bad sided effects, i) the reduction of the life time of the systems and ii) the increase of the failure rate due to hardware faults. Such natural faults demonstrate the need to go beyond classical fault avoidance at development time (usually brought by formal description techniques and properties verification) and to identify all the threats that can impair interactive systems.

Considering human errors

Several contributions in the human factors domain deal with studying internal human processes that may lead to actions that can be perceived as erroneous from an external view point. In the 1970s, Norman, Rasmussen and Reason have proposed theoretical frameworks to analyze human error. Norman, proposed a predictive model for errors [21], where the concept of "slip" is highlighted and causes of error are rooted in improper activation of patterns of action.

Rasmussen proposes a model of human performance which distinguishes three levels: skills, rules and knowledge (SRK model) [25]. This model provides support for reasoning about possible human errors and has been used to classify error types. Reason [26] takes advantages of the contributions of Norman and Rasmussen, and distinguishes three main categories of errors:

1. Skill-based errors are related to the skill level of performance in SRK. These errors can be of one of the 2 following types: a) Slip, or routine error, which is defined as a mismatch between an intention and an action [21]; b) Lapse which is defined as a memory failure that prevents from executing an intended action.
2. Rule-based mistakes are related to the rule level of performance in SRK and are defined as the application of an inappropriate rule or procedure.
3. Knowledge-based errors are related to the knowledge level in SRK and are defined as an inappropriate usage of knowledge, or a lack of knowledge or corrupted knowledge preventing from correctly executing a task.

At the same time, Reason proposed a model of human performance called GEMS [26] (Generic Error Modelling System), which is also based on the SRK model and dedicated to the representation of human error mechanisms. GEMS is a conceptual framework that embeds a detailed description of the potential causes for each error types above. These causes are related to various models of human performance. For example, a perceptual confusion error in GEMS is related to the perceptual processor of the Human Processor model [5].

Causes of errors and their observation are different concepts that should be separated when analyzing user errors. To do so, Hollnagel [15] proposed a terminology based on 2 main concepts: phenotype and genotype. The phenotype of an error is defined as the erroneous action that can be observed. The genotype of the error is defined as the characteristics of the operator that may contribute to the occurrence of an erroneous action.

These concepts and the classifications above provide support for reasoning about human errors and have been widely used to develop approaches to design and evaluate interactive systems [29]. As pointed out in [21] investigating the association between a phenotype and its potential genotypes is very difficult but is an important step in order to assess the error-proneness of an interactive system.

PROPOSALS FOR DEALING WITH SYSTEM FAILURES AND HUMAN ERRORS

Although system failures and human errors can both occur at runtime and be strongly correlated, these two problems are handled separately when developing an interactive system.

Dealing with operational natural faults

The issue of operational natural faults has hardly been studied in the field of human-computer interaction and just a few contributions are available about this topic. However, this issue has long been studied in the field of dependable computing systems. As the operational natural faults are unpredictable and unavoidable, the dedicated approach for dealing with them is fault-tolerance [1] that can be achieved through specialized fault-tolerant architectures, by adding redundancy or diversity using multiple versions of the same software or by fault mitigation: reducing the severity of faults using barriers or healing behaviors [19].

To deal with these faults, we proposed two approaches:

- The reconfiguration of the interaction techniques or possibly the organization of display when required by the occurrence of hardware faults [18].
- The adaptation of fault-tolerant architecture for developing fault-tolerant widgets as proposed in [33] or for extending this approach to all the interactive components of the interactive system (including for example the interaction techniques) as proposed in [10].

Dealing with human errors

Many techniques have been proposed for identifying which human errors may occur in a particular context and what could be their consequences in this given context.

- Several human reliability assessment techniques such as CREAM [12], HEART [35], and THERP [33] are based on task analysis. They provide support to assess the possibility of occurrence of human errors by structuring the analysis around task descriptions. Beyond these commonalities, THERP technique also provides support for assessing the probability of occurrence of human errors.
- Task models based techniques have also been proposed to identify, describe and analyze potential human errors and human tasks deviation such as in [29], [9] and [23].

Dealing with both operational natural faults and human errors

Integrated approaches can be envisioned for taking into account both system faults and human errors. Such approaches can leverage existing techniques in the fields of: dependable computing, human reliability assessment and human computer interaction. As proposed in [16], a stepwise and iterative process can be used to identify in a systematic way human error and system failures for an underdevelopment interactive systems. From this systematic identification, the construction of enriched task models (embedding potential human errors and system faults), can provide support for analyzing their impact and proposing changes for modifying the system.

ILLUSTRATIVE EXAMPLE FROM THE ATM WORLD

The typology of faults introduced in Figure 1 can be easily applied to any application providing support to understand-

ing how the approach followed for the development of a system is addressing the various faults.

In the case of AMAN application proposed for the workshop, the various faults can lead to failures in the management of the aircrafts by the air traffic controllers. For instance, as detailed in [17], we have analysed 3 types of failures leading to 3 automation degradation scenarios: advisories from AMAN being not available anymore, advisories being frozen for a while then starting again and advisories provided being delayed.

If a rigorous development process is followed and formal methods are used (as proposed in DO178-C [7]) one could expect that such failures would not occur. However, natural faults could easily produce such undesired behaviours. Similarly human errors such as not perceiving the advisories or interpreting them incorrectly could also end up with similar malfunction (but this time at organizational level only as the system is supposed to function correctly).

CONCLUSION

This position paper argues that formal methods are good candidates for dealing with development faults. However, this position paper has also presented a typology of faults that identify other sources of failures that development faults: natural faults and human errors.

In order to cover all these faults and to prevent related failures to occur we argued that multiple combined approaches (including formal methods) should be applied. For instance, it is interesting to note that detection and recovering mechanisms for natural faults could be described using formal methods in order to guarantee that their behaviour will be conformant with the expected one (as presented in [34]).

We have not addressed issues related to malicious faults that could however be discussed during the workshop.

REFERENCES

1. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. In IEEE Trans. on Dependable and Secure Computing, vol.1, no.1, pp. 11- 33, Jan.-March 2004.
2. Back J., Blandford A, CurzonP. Recognising Erroneous and Exploratory Interactions. INTERACT (2) 2007: 127-140
3. Barboni, E., Bastide, R., Lacaze, X. Navarre, D., Palanque, P. Petri Net Centered versus User Centered Petri Nets Tools. AWPN 2003 - 10th Workshop on Algorithms and Tools for Petri Nets, Eichstätt, Germany, 26/09/2003-27/09/2003.
4. Beaudouin-Lafon, M. et al. CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets. In Proc. of ICATPN'2001: 22nd International Conference on Application and Theory of Petri Nets (June 2001, Newcastle upon Tyne, England), Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 71-80.
5. Card, S., Moran, T., Newell, A. The model human processor: An engineering model of human performance. John Wiley & Sons, 1986.
6. Dix, A. Upside down As and algorithms – computational formalisms and theory. J. Carroll (Ed.), HCI Models Theories and Frameworks: Toward a Multidisciplinary Science, Morgan Kaufmann, San Francisco (2003), pp. 381–429 (Chapter 14).
7. DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification, published by RTCA and EUROCAE, 2012
8. DO-333 Formal Methods Supplement to DO-178C and DO-278ASoftware Tool Qualification Considerations, published by RTCA and EUROCAE December 13, 2011
9. Fahssi, R., Martinie, C., Palanque, P. Enhanced Task Modelling for Systematic Identification and Explicit Representation of Human Errors. In Proc. of IFIP TC 13 Intl. Conf. on HCI, INTERACT 2015, Bamberg.
10. Fayollas C., Fabre J-C., Palanque P., Barboni E., Navarre D., Deleris Y: Interactive cockpits as critical applications: a model-based and a fault-tolerant approach. Int. Journal on Critical Component-Based Software 4(3): 202-226 (2013)
11. Fayollas, C., et al. A Software-Implemented Fault-Tolerance Approach for Control and Display Systems in Avionics. In Proc. PRDC 2014, IEEE, 21-30.
12. Gram, C., Cockton, G. Design principles for Interactive Software. London. 1996. Chapman & Hall.
13. Hecht H. and Fiorentino E. Reliability assessment of spacecraft electronics. In Annual Reliability and Maintainability Symp., pages 341–346. IEEE, 1987.
14. Hollnagel E. "The phenotype of erroneous actions Implications for HCI design," in G. R. S. Weir and J. L. Alty, Eds , Human Computer Interaction and the Complex Systems, London: Academic Press, 1991.
15. Hollnagel, Erik. Cognitive reliability and error analysis method (CREAM). Elsevier, 1998.
16. Martinie C., Palanque P., Ragosta M., Sujan M-A., Navarre D., Pasquini A.: Understanding Functional Resonance through a Federation of Models: Preliminary Findings of an Avionics Case Study. SAFECOMP 2013: 216-227
17. Martinie, C., Palanque, P., Fahssi, R., Blanquart, J.-P., Fayollas, C., Seguin, C. Task Model-Based Systematic Analysis of Both System Failures and Human Errors. IEEE Transactions on Human-Machine Systems, to appear in 2015.
18. Navarre, D., Palanque, P. and S. Basnyat, "A formal approach for user interaction reconfiguration of safety critical interactive systems," in Proc. Int. Conf Comp. Safety, Rel. Security, 2008, pp. 373–386.

19. Neema, S., Bapty, T., Shetty, S., and Nordstrom; S. Autonomic fault mitigation in embedded systems. Eng. Appl. Artif. Intell., vol. 17, no. 7,pp. 711–725, 2004.
20. Nicolescu B., Peronnard P., Velazco R., and Savaria Y. Efficiency of Transient Bit-Flips Detection by Software Means: A Complete Study. Proc. of the 18th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT '03). IEEE Computer Society, 377-384.
21. Norman, D. A. (1981). Categorization of action slips. Psychological review, 88(1), 1.
22. Papatzanis, G, Curzon, P., Blandford, A. Identifying Phenotypes and Genotypes: A Case Study Evaluating an In-Car Navigation System. EHCI/DS-VIS 2007, pp. 227-242.
23. Paterno, F., Santoro, C., "Preventing user errors by systematic analysis of deviations from the system task model", 2002, *Int. Journal on Human Computing Systems*, Elsevier, vol. 56, n. 2,pp. 225-245.
24. Polet, P, Vanderhaegen, F, and Wieringa, P. Theory of safety related violation of system barriers. Cognition Technology & Work, 4, 3, 171-179. 2002.
25. Rasmussen, J. Skills, rules, knowledge: signals, signs and symbols and other distinctions in human performance models, IEEE transactions: Systems, Man &Cybernetics, 1983.
26. Reason, J T. Generic error modelling system: a cognitive framework for locating common human error forms. New technology and human error, 63, 86. 1987.
27. Reason, J. (1990). Human Error, Cambridge University Press
28. Regis, D.; Hubert, G.; Bayle, F.; Gatti, M., "IC components reliability concerns for avionics end-users," Digital Avionics Systems Conference IEEE/AIAA 32nd pp.2C2-1,2C2-9, 5-10 Oct. 2013
29. Rimvydas Ruksenas, Paul Curzon, Ann Blandford, Jonathan Back: Combining Human Error Verification and Timing Analysis. EHCI/DS-VIS 2007: 18-35
30. Rizzo, A., Ferrante, D., & Bagnara, S. (1995). Handling human error. Expertise and technology: Cognition & human-computer cooperation, 195-212.
31. Schroeder B., E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In ACM SIGMETRICS, pages 193–204, Seattle, WA, June 2009.
32. Silva, J-L., Fayollas, C., Hamon, A., Palanque, P., Martinie, C., Barboni, E. Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification. International Workshop on Formal Methods for Interactive Systems (FMIS 2013), London, 24/06/2013, Electronic Communications of the EASST.
33. Swain, A. D., Guttman, H. E. Handbook of Human Reliability Analysis with Emphasis on Nuclear Power Plant Applications, Final report. NUREG/CR- 1278. SAND80-0200. RX, AN. US Nuclear Regulatory Commission, August 1983.
34. Tankeu-Choitat, A. et al. Self-checking components for dependable interactive cockpits using formal description techniques. In Proc. PRDC 2011, 164-173.
35. Williams, J. C. A data-based method for assessing and reducing human error to improve operational performance. In Human Factors and Power Plants, IEEE, 1988. p. 436-450.

A User-Centered View on Formal Methods: Interactive Support for Validation and Verification

Eric Barboni, Arnaud Hamon, Célia Martinie & Philippe Palanque

ICS-IRIT, University of Toulouse,
118 Route de Narbonne,
F-31062, Toulouse, France
{Name}@ irit.fr

ABSTRACT

During early phases of the development of an interactive system, future system properties are identified (through interaction with end users e.g. in the brainstorming and prototyping phases of the development process, or by requirements provided by other stakeholders) imposing requirements on the final system. Some of these properties rely on informal aspects of the system (e.g. satisfaction of users) and can be checked by questionnaires, while other ones require the use of formal methods. Whether these properties are specific to the application under development or generic to a class of applications, the verification of the presence of these properties in the system under construction usually involve verification tools to process the formal description of the system. The usability [26] of these tools has a significant impact on the V&V phases which usually remains perceived as very resource consuming. This position paper proposes the application of action theory to identify complex aspects of verification and exploits it for identifying areas of improvement.

Author Keywords

Formal methods; interactive systems; object oriented Petri nets; Analysis.

ACM Classification Keywords

D.2.2 [Software] Design Tools and Techniques – Computer aided software engineering (CASE).

INTRODUCTION

Nowadays interactive applications are deployed in more and more complex command and control systems including safety critical ones. Dedicated formalisms, processes and tools are thus required to bring together various properties such as reliability, dependability and operability. In addition to standard properties of computer systems (such as safety or liveness), interaction properties have been identified. Properties related to the usage of an interactive system are called external properties [6] and characterize the capacity of the system to provide support for its users to accomplish their tasks and goals, and prevent or help to recover from errors. Although all types of properties are not always completely independent one from each other (any might be conflicting), external properties are related to the user's point of view and usability factor, whereas internal properties are related to the design and development process of the system itself (modifiability, run time efficiency). Interactive systems have to support both types of properties and dedicated techniques and approaches have been studied for this purpose, amongst them are formal methods. Formal languages have proven their value in several domains and are a necessary condition to understand, design, develop systems and check their properties.

Formal methods have been studied within the field of HCI as a means to analyze in a complete and unambiguous way interactions between a user and a system. Several types of approaches have been developed [9], which encompass contributions about formal description of an interactive system and/or formal verification of its properties. Other approaches such as [6] exploit formal methods for understanding interactive systems and provide a better description of their specificities.

The use of formal methods depends on the phase of the development process describing the activities necessary to develop the interactive system under consideration. In the case of critical interactive systems, [16] proposes a development process relying on formal methods and taking into account both interactive and critical aspects of such systems which necessitates several formal descriptions used by different user types (system designers, human factor specialists…). Consequently, the usability of tools for validation and verification of these interactive systems target will depend on their users' activity in the development process.

Whether being used as a means for describing in a complete and unambiguous way the interactive system or as means for verifying properties, formal description techniques and their associated tools need to be designed to be usable and not error prone.

To cite this article use the following information:

Eric Barboni, Arnaud Hamon, Célia Martinie, and Philippe Palanque. 2015. A User-Centered View on Formal Methods: Interactive Support for Validation and Verification. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 24-29.
http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

The paper proposes a user–centered view on the use of formal methods. We take as a running example the ICO notation [18] and its associated tool Petshop [2] and use Norman's action theory [19] as a framework for identifying usability gaps and error sources. We believe however that this framework would be applicable to other notations and tools and hope to discuss this during the workshop.

VALIDATION AND VERIFICATION TOOLS FOR THE DEVELOPMENT OF INTERACTIVE CRITICAL? SYSTEMS

Norman's action theory is quickly presented in Figure 1. Its application to formal modelling is presented in Figure 2.

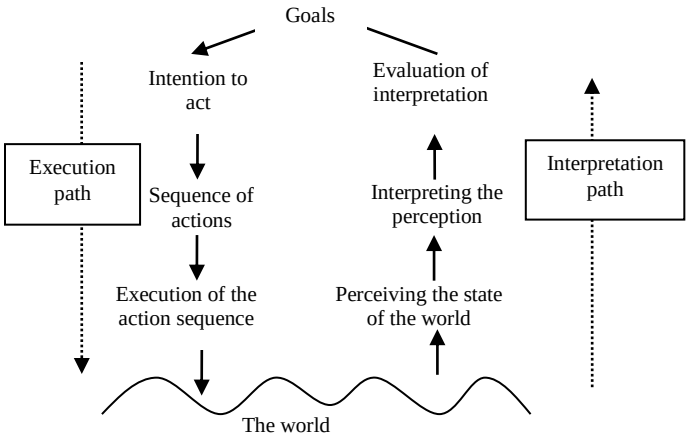

Figure 1. The seven stages of the action theory

That figure shows the refinement of specific activities to take into account modelling activities. The two main stages we are focusing on are:

- On the execution side, the activity of going from an intention to its actual transcription into some model,
- On the perception side, the activity of perceiving model behavior and interpreting this perception.

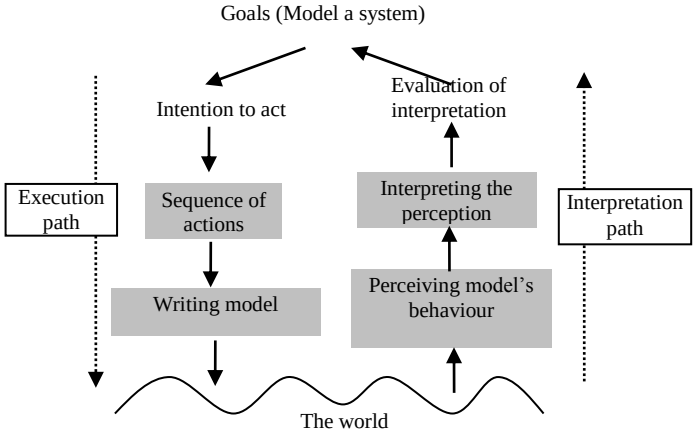

Figure 2. Formal modelling within the action theory

User tasks when validating and verifying interactive critical systems

We have identified three main tasks when verifying formal model, and Petri nets (which is the underlying formalism of ICOs) in particular: analysis, simulation in conjunction of the edition task the formers reflecting the verifiability [11] and the latter the executability [8] aspects.

Simulation is the task where users (people building the formal model) have to check that the model built exhibits the expected behavior. The interpretation task can be eased if the state changes in the model are shown in an animated way thus reducing the users' activity of comparing the current state with the previous one.

For analysis related tasks, users check the validity of some properties hold on the model. One of the issues is then to understand the analysis result (for instance one place in not in any "P invariant") in Petri net tools and then to map this analysis result with the goal (was this place meant to be unbounded?).

Existing tools for validation and verification of interactive critical systems

Verification and validation of formal models can be divided in two categories, whether they rely on static analysis or on simulation (step by step, interactive…).

System models

A representative classification of User Interface Description Languages (UIDLs) have shown only few frameworks describing the interactive system behavior and providing support for analysis [10].

Among these, Marigold [24] addresses limited validation and verification analysis based on reachability graph analysis and allows exporting to the Integrated Net Analyzer tool [22] offering other analysis capabilities. Proton++ [14] only provides static analysis as the tool only handles gesture conflict detection between models at compilation time. ICO [18] provides support for validation and verification but only through invariant analysis (Place/Transition invariants) provided by the Petshop tool as detailed in the tool description section.

Finally, the colored Petri nets (CPN) approach is more complete in terms of analysis enabling validation, verification and performance analysis accomplished by all different types of analysis techniques (e.g. reachability analysis) except invariants. The analysis of CPN models is supported by the CPN Tools [13].

In addition, both CPN and PetShop enable simulation of the models they produce. However, CPN does not connect the model to the user interface of the application requiring an additional step when interactive application are concerned, forcing the system designers .

Tasks models

CTT [21] proposed an approach based on formal model-checking (with CADP[1] toolset) of LOTOS [12] specifications of tasks between the user and the system. The HAMSTERS tool [15] (supporting the HAMSTERS notation) also provides static analysis support via basic check on the model's structure.

However, here again, tasks have to be checked on the user interface and simulation should integrate execution on the user interface. If this is not the case then a gulf exists between the task model and its simulation and the actual behavior of the interactive application.

Tools Usability

Building or modifying system and task models belongs to the type of human activities that is highly demanding on the user's side. Figure 2 provides a generic framework for investigating where the main difficulties can occur, and thus to provide design rules for environments to support user's activities and reduce difficulties.

In order to increase usability during validation and verification phases, tools should provide "continuous and permanent feedback", "modeless support to users' tasks", "reversibility of actions (undoing actions)" and taking into account the different formalisms' characteristics. This dimension becomes particularly important when considering large and complex interactive systems for which both user and system models are used jointly.

CIRCUS: A CASE TOOL FOR FORMAL ANALYSIS AND DESCRIPTION OF INTERACTIVE SYSTEMS

In the following section, we illustrate the validation and verification of interactive system using formal methods with the Circus suite which combines task models in HAMSTERS and high level Petri nets in

Notations supported by the tool suite

The ICO notation

The ICO notation [18] (Interactive Cooperative Objects) is a formal description technique devoted to specify interactive systems. Using high-level Petri nets for dynamic behavior description, the notation also relies on object-oriented approach (dynamic instantiation, classification, encapsulation, inheritance and client/server relationships) to describe the structural or static aspects of systems.

This notation have been applied to formally specify interactive systems in the fields of Air Traffic Management [18], satellite ground systems [20] and cockpits of military [4] and civil [1] aircrafts.

The HAMSTERS' tool tasks notation

HAMSTERS features a task model notation that enables structuring users 'goals and sub-goals into a hierarchical tasks tree in which qualitative temporal relationship amongst tasks are described by operators [17]. Goals or sub-goals are modeled using the type of task called "abstract". An abstract task can be refined in 3 types of tasks: "user task", "system tasks" and "interactive tasks". A "user task" can be refined in the following sub-types: "perceptive task", "cognitive task" and "motor task". An interactive task can be refined in the following sub-types: "input task" and "output task".

In addition, [25] extended the notation to integrate objects, knowledge and information; thus describing the exchanges between users and the systems they interact with.

Tool description

The following sub-sections describe both tools (system and task models' tool) which are merged into a single framework called Circus as introduced in [2].

Though one recommendation is to provide modeless tools, PetShop combines these three modes and allow the users direct visualization of the structural analysis results while editing and simulating the Petri nets' models. These analysis mode can be activated without stopping either the edition or the simulation. In addition, a dedicated panel presents the incidence matrix and the P/T invariants, which allow the identification of potential deadlock in the system models.

The HAMSTERS tool [15] is the part of the framework which enable task model editing and analysis.
The Circus tool provides a framework for both system and task models. It enables the co-execution of the system with the corresponding user's task model [2] which corresponds to executability related tasks. Regarding the verifiability, the tool provides a checking mechanism to ensure tasks and system models are compatible.

Tool usability

The Circus tool targets engineers, system designers and human factors specialist and helps them achieve their specific tasks while developing interactive critical systems. Their objective is to design and develop usable, reliable and dependable applications for these critical systems. It encompasses formal verification of the system's behavior as well as its compatibility with the system's targeted users.
The Circus tool development has been, so far, oriented towards enhancing one particular dimension of usability: effectiveness. We developed new capabilities so the tools' users can achieved their goals by being able to complete their tasks during most phases of the development process. Therefore, the tool enables the analyst to formally validate both system and tasks models and ensure they match so that the end-users will be able to perform their tasks on the system described. In addition, the tool allows the assessment of

[1] http://cadp.inria.fr/

the impact of dependability on usability on the considered interactive critical systems as well as exhibiting design choices and trade-offs in (potentially) conflicting user interface guidelines. Finally Circus provides support for usability testing by providing execution logs at the model level so system designers are able to match the potential issues with the model's nodes and opens the way to user performance measurement.

Although the various Circus user types are identified, we did not measure their satisfaction and analyze improvements to be made with respect to their user profiles and the related tool functionalities. The evaluation of users' efficiency during verification and validation is also identified for future work with performance measures among others.

ILLUSTRATIVE EXAMPLE: WXR APPLICATION

This section illustrates the use of the Circus tool which enables the formal description of interactive critical system as a socio-technical system; and which supports is validation and verification.

Case study description

Weather radar (WXR) is an application currently deployed in many cockpits of commercial aircrafts. It provides support to pilots' activities by increasing their awareness of meteorological phenomena during the flight journey, allowing them to determine if they may have to request a trajectory change, in order to avoid storms or precipitations for example.

Figure 3 - Image of a) the weather radar control panel b) of the radar display manipulation

Figure 3 presents a screenshot of the weather radar control panel, used to operate the weather radar application. This panel provides two functionalities to the crew. The first one is dedicated to the mode selection of weather radar and provides information about status of the radar, in order to ensure that the weather radar can be set up correctly. The operation of changing from one mode to another can be performed in the upper part of the panel.

The second functionality, available in the lower part of the window, is dedicated to the adjustment of the weather radar orientation (Tilt angle). This can be done in an automatic way or manually (Auto/manual buttons). Additionally, a stabilization function aims to keep the radar beam stable even in case of turbulences. The right-hand part of Figure 3 presents an image of the controls used to configure radar display, particularly to set up the range scale (right-hand side knob with ranges 20, 40, … nautical miles).

Figure 4 shows screenshots of weather radar displays according to two different range scales (40 NM for the left display and 80 NM for the right display). Spots in the middle of the images show the current position, importance and size of the clouds.

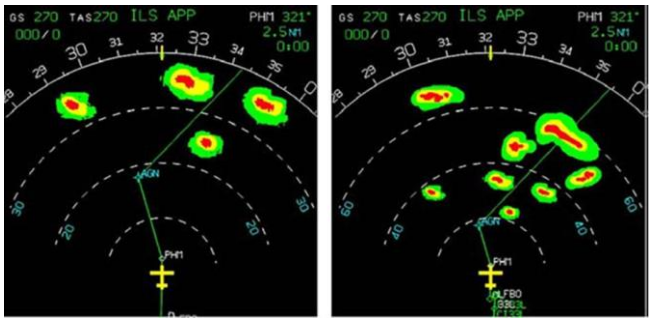

Figure 4 - Screenshot of weather radar displays

Formal modelling of the application's behavior using ICO

The first model presented here describes how it is possible to handle the weather radar configuration of both its mode and its tilt angle.

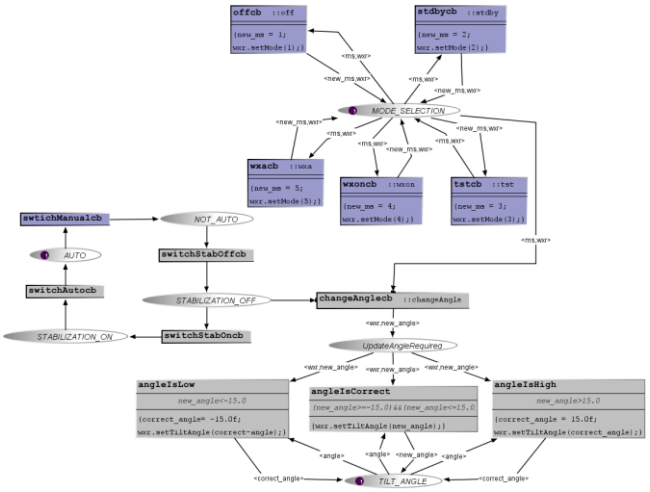

Figure 5 - Behavior of the WXR mode selection and tilt angle setting

Figure 3 shows the interactive means provided to the user to:

- Switch between the five available modes (upper part of the figure) using radio buttons (the five modes being WXON to activate the weather radar detection, OFF to switch it off, TST to trigger a hardware checkup, STDBY to switch it on for test only and WXA to focus detection on alerts).

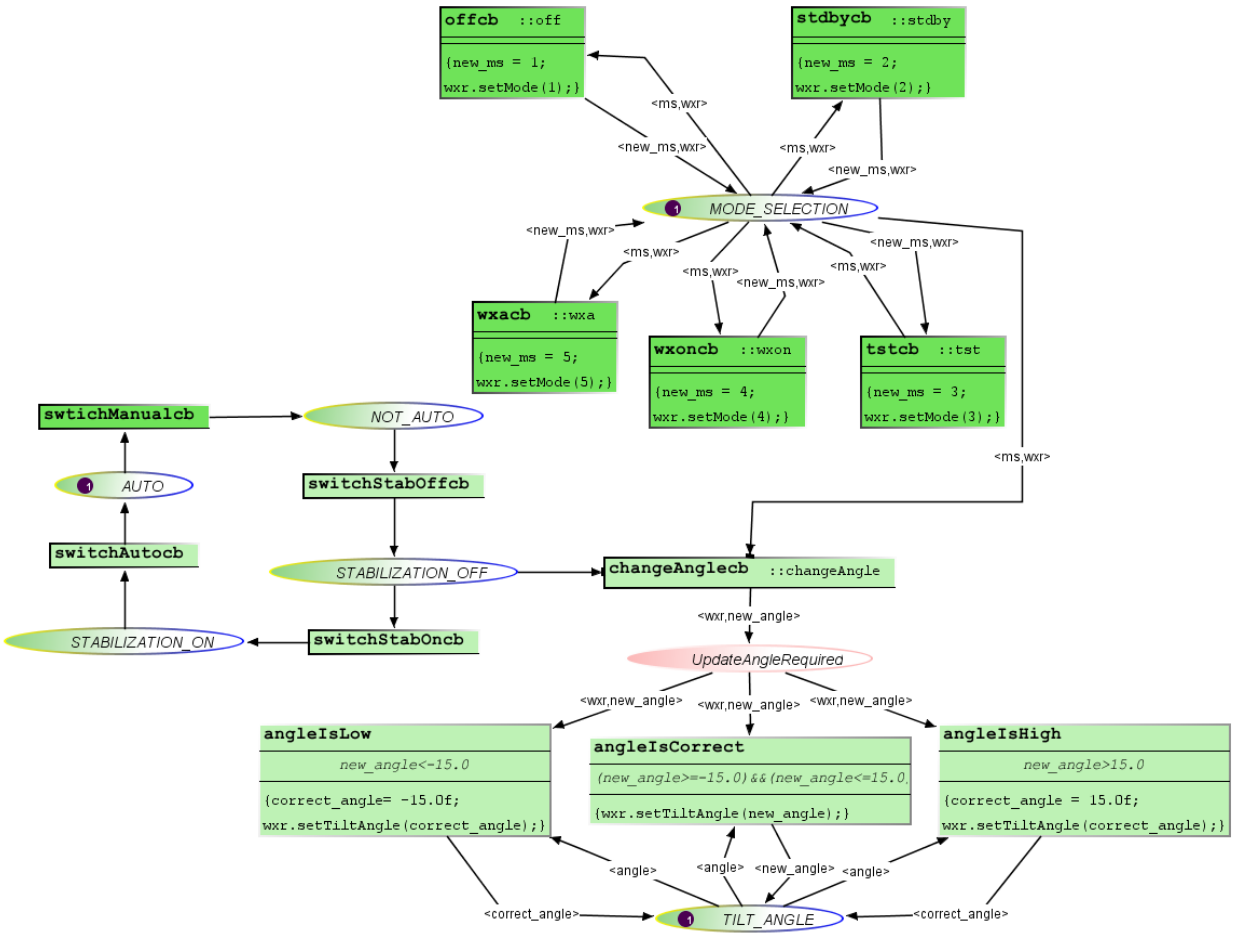

Figure 6 - Representation of invariants with the Analysis feature of Petshop CASE tool

- Select the tilt angle control mode (lower part of the figure) amongst three modes (fully automatic, manual with automatic stabilization and manual selection of the tilt angle.

The corresponding task model is not presented in this paper but is described in [15].

Illustration of verification and validation tasks to be tool supported

In this section we emphasis on the validation aspect of the system model presented in Figure 5. When activated, the static analysis mode of PetShop displays a dedicated panel we do not present in this paper but which results are also displayed in the main edition view as shown Figure 6. The green overlay on the places and transitions identifies the node part of the model's invariants otherwise the red overlay is used as for the place UpdateAngleRequired. Places with yellow borders are syphons whereas taps use a blue stroke.

In order to determine which nodes belong to the same invariant as another node, a model pop-up menu is provided, taking over the standard pop-up menu for edition. This modal pop-up menu can be switch from analysis mode to normal edition mode using a dedicated icon on the toolbar.

The main current limitation of the tool regarding the static analysis lies in the fact that the PetShop algorithms do not close opened models (models that provided services to other models).

CONCLUSION AND PERSPECTIVES

This position paper has presented the use of a generic model in the field of HCI and its application for identify issues related to the use of formal methods for interactive systems.

The framework has been applied to the tool suite called CIRCUS which embeds the HAMSTERS and Petshop tools.

We would like to discuss and possibly extend and refine this approach to understand better where usability problems arise while using formal methods for the design and analysis of interactive systems.

We hope also that participants will provide information about the notations and tools they are using to assess if the proposed framework is applicable more widely.

REFERENCES

1. Barboni E., Conversy S., Navarre D. & Palanque P. Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. 13th conf. on Design Specification and Verification of Interactive Systems (DSVIS 2006), LNCS Springer Verlag. p25-38
2. Barboni E., Ladry J.-F., Navarre D., Palanque P., and Winckler M.. 2010. Beyond modelling: an integrated environment supporting co-execution of tasks and systems models. In Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems (EICS '10). ACM, New York, NY, USA, 165-174.
3. Barboni, E., Bastide, R., Lacaze, X. Navarre, D., Palanque, P. Petri Net Centered versus User Centered Petri Nets Tools. AWPN 2003 - 10th Workshop on Algorithms and Tools for Petri Nets, Eichstätt, Germany, 26/09/2003-27/09/2003.
4. Bastide R., Navarre D., Palanque P., Schyn A. & Dragicevic P. A Model-Based Approach for Real-Time Embedded Multimodal Systems in Military Aircrafts. Sixth International Conference on Multimodal Interfaces (ICMI'04) October 14-15, 2004, USA, ACM Press
5. Beaudouin-Lafon, M. et al. CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets. In Proc. of ICATPN'2001: 22nd International Conference on Application and Theory of Petri Nets (June 2001, Newcastle upon Tyne, England), Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 71-80.
6. Dix, A. Upside down ∀s and algorithms – computational formalisms and theory. J. Carroll (Ed.), HCI Models Theories and Frameworks: Toward a Multidisciplinary Science, Morgan Kaufmann, San Francisco (2003), pp. 381–429 (Chapter 14).
7. Dix, A. Formal methods for interactive systems. Academic Press, 1991.
8. Fuchs, N.E. Specifications are (preferably) executable. Journal on Software Engineering, vol. 7, issue 5, September 1992, pp. 323-334.
9. Gram, C., Cockton, G. Design principles for Interactive Software. London. 1996. Chapman & Hall.
10. Hamon A., Palanque P., Silva J-L., Deleris Y., and Barboni E. 2013. Formal description of multi-touch interactions. In Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems (EICS '13). ACM, New York, NY, USA, 207-216.
11. Hayes, I., Jones, C.B. Specifications are not (necessarily) executable. Journal on Software Engineering, vol. 4, issue 6, November 1989, pp. 330-338.
12. International Standard Organisation, ISO 8807:1989, Information processing systems -- Open Systems Interconnection -- LOTOS -- A formal description technique based on the temporal ordering of observational behaviour, 1989.
13. Jensen, K., Kristensen, L. M., & Wells, L. (2007). Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Intern. Journ. on Software Tools for Technology Transfer, 9(3-4), 213-254
14. Kin K., Hartmann B., DeRose T., and Agrawala M.. 2012. Proton++: a customizable declarative multitouch framework. In Proc. of the 25th annual ACM Symp. on User Interface Software and Technology (UIST '12). ACM, 477-486
15. Martinie C., Palanque P., Barboni E., Winckler M., Ragosta M., Pasquini A., and Lanzi P.. 2011. Formal tasks and systems models as a tool for specifying and assessing automation designs. In Proceedings of the 1st International Conference on Application and Theory of Automation in Command and Control Systems (ATACCS '11). IRIT Press, France, 50-59.
16. Martinie C., Palanque P., Navarre D., and Barboni E.. 2012. A development process for usable large scale interactive critical systems: application to satellite ground segments. In Proceedings of the 4th international conference on Human-Centered Software Engineering (HCSE'12), Springer-Verlag, Berlin, Heidelberg, 72-93.
17. Martinie C., Palanque P., Winckler M.. Structuring and Composition Mechanisms to Address Scalability Issues in Task Models. IFIP TC13 Human Computer Interaction 2011 (INTERACT). p:134-152. Springer-Verlag.
18. Navarre D., Palanque P., Ladry J-F. & Barboni E. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans. Comput.-Hum. Interact., 16(4), 18:1–18:56. 2009
19. Norman, D. «The Design of Everyday Things (Originally published: The psychology of everyday things).» The Design of Everyday Things (Originally published: The psychology of everyday things). New York: Basic Books, 1988.
20. Palanque P., Bernhaupt R., Navarre D., Ould M. & Winckler M. Supporting Usability Evaluation of Multimodal Man-Machine Interfaces for Space Ground Segment Applications Using Petri net Based Formal Specification. Ninth Int. Conference on Space Operations, Italy, June 18-22, 2006
21. Paternó, F., Santoro, C. Integrating model checking and HCI tools to help designers verify user interface properties. Palanque and Paternó (eds), DSV-IS 2000 Interactive Systems: Design, Specification and Verification. LNCS 1946, Springer 2001, pp. 135–150.
22. Roch, S. and P. H. Starke (1999, April). INA Integrated Net Analyser (V. 2.2). Humboldt-Universitat zu Berlin.
23. Silva, J-L., Fayollas, C., Hamon, A., Palanque, P., Martinie, C., Barboni, E. Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification. International Workshop on Formal Methods for Interactive Systems (FMIS 2013), London, 24/06/2013, Electronic Communications of the EASST.
24. Willans, J. S. and Harrison, M. D. 2001. Prototyping pre-implementation designs of virtual environment behavior. In Proc. of the 8th IFIP Int. Conf. on Engineering for Hum.-Comput. Interact., Lecture Notes In Comput. Sc., vol. 2254. Springer, 91–10.
25. Martinie C., Palanque P., Ragosta M., and Fahssi R.. 2013. Extending procedural task models by systematic explicit integration of objects, knowledge and information. In Proceedings of the 31st European Conference on Cognitive Ergonomics (ECCE '13). ACM, New York, NY, USA, Article 23, 10 pages.
26. International Standard Organization. (1996). DIS 9241-11: Ergonomic requirements for office work with visual display terminals (VDT) - Part 11 Guidance on Usability.

The LIDL Interaction Description Language

Vincent Lecrubier
ONERA DTIM/LAPS
Toulouse, France
lecrubier@onera.fr

Bruno d'Ausbourg
ONERA DTIM/LAPS
Toulouse, France
ausbourg@onera.fr

Yamine Aït-Ameur
ENSEEIHT
Toulouse, France
yamine@enseeiht.fr

ABSTRACT

This paper describes LIDL, a language dedicated to the specification of interactive systems. LIDL is based on the idea that most programming languages are useful to specify computations, but are not adequate when it comes to specifying interactions. We first introduce the context and the need for new paradigms for interactive systems specification. Then we describe the basic concepts of LIDL, such as Interfaces, Data activation, Interactions, and LIDL program structure. Some uses of LIDL programs such as verification and code generation are then explained. Finally, a boiling water nuclear reactor user interface is partially developed using LIDL, as an example use case.

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords

Human-Machine Interfaces; Domain-Specific Language; Interactive Systems; Formal Language; Critical Systems

INTRODUCTION

A lot of research work have focused on how to design, program and verify functional concerns for critical systems and more particularly aeronautical systems. HMI systems did not benefit from the same attention and efforts.

A significant amount of work has focused on devising models for the development process of software systems in the field of software engineering.

The system development process in critical domains as, for instance, in aeronautics inherited these models. This process is now widely based on the use of standards that take into account the safety and security requirements of the systems under construction. In particular the DO178C standard [1], in aeronautics, defines very strict rules and instructions that must be followed to produce software products, embedded systems and their equipments. The objective is to ensure that the software performs its function with a safety level in accordance with the safety requirements.

To cite this article use the following information:

Vincent Lecrubier, Bruno d'Ausbourg, and Yamine Aït-Ameur. 2015. The LIDL Interaction Description Language. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 30-37. http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

Because of the problem stated in Figure 1, the HMI development does not follow the same processes. Nevertheless, in aeronautics, HMI systems are now made up by multiple hardware and software components embedded in aircraft cockpits. These systems are large and complex artifacts that also face tough constraints in terms of usability, security and safety. They support interactive applications that must behave as intended with a high degree of assurance because of their criticity. An error in the software components that implement interactions in these applications may lead to a human or system fault that may have catastrophic effects.

For example, the BEA report [6] about the crash of Rio-Paris AF-447 A330 Airbus establishes that, during the flight, interface system displayed some actions to be performed by the pilot in order to change the pitch of the aircraft and to nose it up while it was stalling. These indications should clearly not have been displayed. Indeed, by following those erroneous displayed instructions the pilot increased the stalling of the aircraft.

In fact, in the industrial context, the development process of critical interactive embedded applications stays very primitive. The usual notations are essentially textual and coding is generally performed from scratch or by reusing previous developments based themselves on textual specifications. In aeronautics, the produced code must be in conformance with the ARINC 661 standard [4]. It may be noticed that some tools recently appeared to enhance the design and coding stages of these systems. But these tools, as for instance Scade Display [23], deal mainly with presentation layers of the systems and do not deal with their complex functional behaviour. In this context, the validation process of the interactive applications is very restricted and poor because it resides practically only in a massive test effort and in expensive evaluation phases at the end of the development process. Moreover there is no actual formal reference to check the implementation is in conformance with. So new approaches and new paradigms are today needed to help in the development process of critical interactive embedded applications.

THE LIDL LANGUAGE

It seems to us that the current state of the art provides no complete solution to the need described in the previous section. The aim of LIDL is to provide a language and tools to deal with this set of problems.

Informal presentation

While most programming languages focus on the description of *computations*, the main idea behind LIDL is to describe

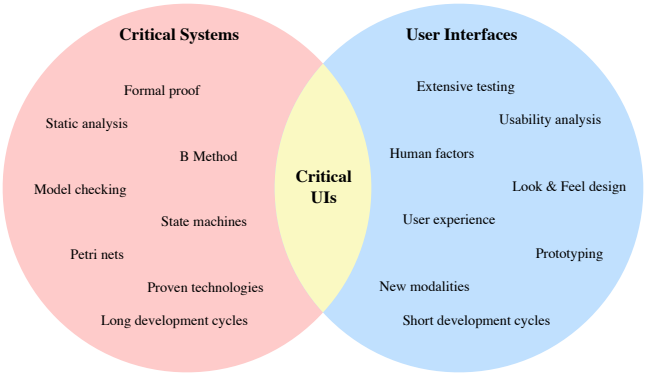

Figure 1. Critial UIs development is as the intersection of two clashing domains.

interactions. This is quite a paradigm shift in the sense that many experienced programmers will at first be surprised by the language semantics. However, we argue that LIDL provides an easier way to specify interactive systems, since its main concepts (interfaces and interactions) are more relevant to the field of interactive systems than other programming languages concepts (objects, functions, algorithms...)

LIDL programs are defined in a declarative manner, and represent interactive systems whose execution is synchronous.

Interfaces

Typed programming languages rely on data types to check the composability of functions and operations. This is convenient when the goal is to describe *computations*. But this is not enough when we try to describe *interactions*. When composing interactions, another very important aspect which is rarely stated is the *direction* data goes in.

As an answer to that matter, an important feature of LIDL is the notion of *interface*. An interface is the combination of two orthogonal aspects: the data type and the data direction.

The notion of data type is well known to most programmers. The notion of data direction is also quite easy to understand: the data can either go *in* or go *out*. The notion of interface is hence quite easy to catch, here are a few example of basic interfaces: `Number in`, `Boolean out`, `Text in`...

The same way compound data types exist, one can express compound interfaces. The syntax to specify compound interfaces is inspired by the Javascript Object Notation (JSON) [14]. Listing 1 shows an example compound interface defined in LIDL.

```
interface Example is
  {
    redSquares      : Square in,
    greenPentagons  : Pentagon in,
    yellowTriangles : Triangle out,
    blueCylinders   : Cylinder out
  }
```

Listing 1. LIDL definition of the example interface

Metaphorically, interfaces can be seen as the specification of pipes of specific shapes that allow objects to go in specific directions. Figure 2 shows a way to visualise the example interface of Listing 1.

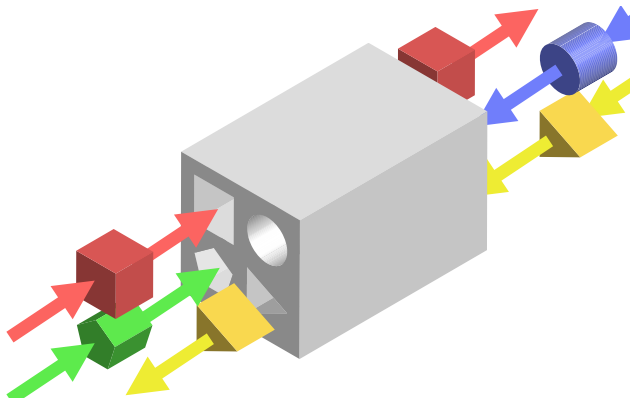

Figure 2. A metaphor of interfaces as pipes that allow specific data types to flow in specific directions

Every interface has a conjugate interface, which has the same data types, but opposite directions. Two interactive systems can only connect if their interfaces are conjugate. This is the consequence of the natural intuition that the output of an entity is the input of another one.

Interfaces are central in LIDL as they have the same role as data types in typed programming languages.

Data activation

Interactive systems rely on two different paradigms: flow-based representations and event-based representations.

Flow-based representations maps well to systems whose data is defined on *continuous* time intervals, such as the pressure inside a reactor. Examples of flow-based representations include Lustre [17], Scade...

On the other hand, event-based systems maps well to systems whose data is defined on discrete time sets, such as clicks on a button. Examples of event-based representations include most User Interface (UI) Toolkits such as Java Swing, Qt...

Several approaches tried to bridge the gap between flow and event representations [2]. However most approaches are biased toward one paradigm or the other. Interestingly, some approaches treat input and output differently, for example by only allowing discrete inputs (events) and continuous output (status). Figure 3 presents the positioning of different academic approaches regarding this aspect. Shown approaches include [17], [12], [11], [13], [18], [3], [21], [20] and LIDL.

Restriction to a paradigm or the other often prevents natural description of interactive systems, which generally are best described using a mix of both. LIDL proposes a simple way to unify and mix the two paradigms: the notion of data activation.

The notion of data activation is latent in industrial art. Most languages exhibit constructions such as the *null* value, the *maybe* monad, callback functions, listeners, observers, signal slots...

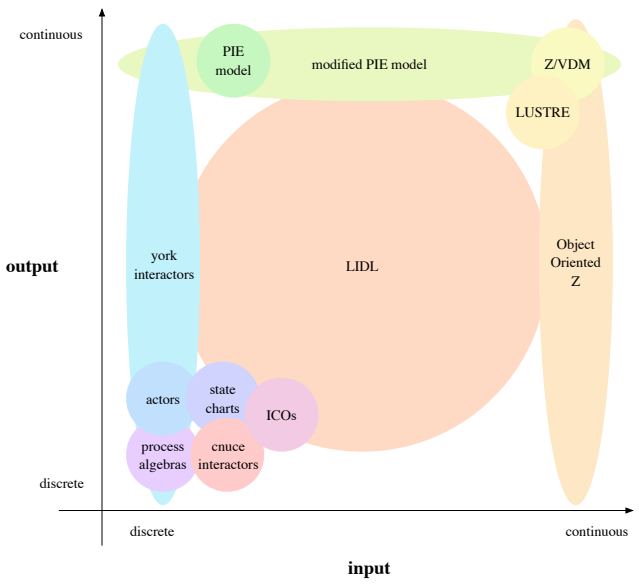

Figure 3. Positions of different academic approaches in the flow vs event space.

In the context of interactive systems, all these constructions boil down to one unique concept: identify the presence of a piece of data, most of the time a message that has to be received or sent. This is exactly what the data activation feature of LIDL does.

Without exception, every piece of data in a LIDL program integrates a notion of activation. The implementation is really simple: *all* LIDL data types are extended with the *inactive* value noted $\perp$. For example, the following table shows example values for the basic data types of LIDL:

Type	Example values
`Activation`	$\perp, \top$
`Boolean`	$\perp$, *true*, *false*
`Number`	$\perp$, 0, 1, 3.14159
`Text`	$\perp$, "*Foo*", "*Bar*", "*Baz*"

Very simplistically, a flow is represented in LIDL by a piece of data which is almost always active. For example, through an execution, the pressure in a reactor would have the following trace: $\{451, 453, 452, 450, 454, ...\}$. On the other hand an event is represented by a piece of data which is almost always inactive. For example, through an execution, clicks on a button would have the following trace: $\{\perp, \perp, \perp, click, \perp, ...\}$

The notion of activation does not break composability. Here is a compound data type expressed in LIDL : `{x:Number,y:Number}`. This data type is a labelled product data type, similar to a `struct` of the C language. Here are a few example of values of this type: `{x:3,y:2}`, `{x:⊥,y:3}`, $\perp$.

Interactions

LIDL is a language to describe interactions. The interaction language has a simple syntax, which uses a lot of parentheses. An interaction is a phrase between parentheses, and it composes trivially. Listing 2 shows an example interaction expression, while Figure 4 shows its structure.

```
1 (when(not(powered))then(turn(light)red))
```

Listing 2. An example interaction expression

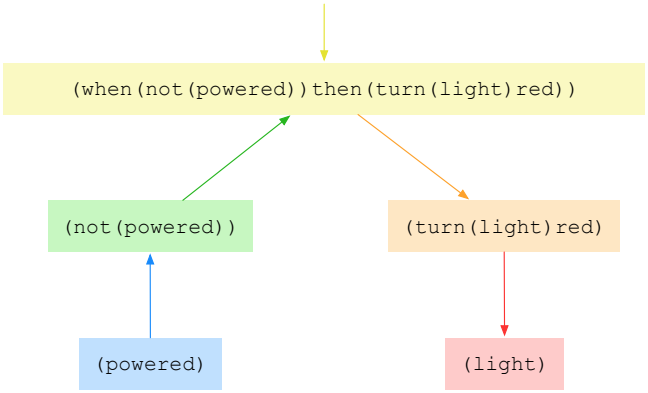

Figure 4. The structure of the example expression. Arrows represent the data flow direction

The semantics of interactions is the most challenging part of LIDL for newcomers, because it is the most disruptive part of the language, since it leverages interfaces and the notion of activation.

Each interaction (i.e. each pair of parentheses, i.e. each block in Figure 4) is attributed a value at each execution step. Depending of the data direction of the interface of the interaction, this value can be defined by the interaction itself (**`out`**) or by an external interaction (**`in`**).

Interactions that comply to the **`out`** interface behave like functions. They output a value, based on their arguments. For example `(not(powered))` receives a boolean `(powered)` and outputs a boolean that is the negation of `(powered)`. This is explained in the following table, which should be easy to understand, with parameters on the left column, and results on the right:

`(powered)`	`(not(powered))`
true	*false*
false	*true*
$\perp$	$\perp$

Interactions that comply to the **`in`** interface behave the opposite way, which is **completely foreign** to programmers. Imagine a function that does not *return* a value based on the arguments it receives, but that *receive* a value, and returns values to its arguments. For example `(turn(light)red)` receives an activation, and outputs a colour `light` which is red when the interaction is active, or $\perp$ the rest of the time. This is summarised in the following table, which will look **unfamiliar** to most programmers, with parameters on the left column, and computation result on the right:

`(turn(light)red)`	`(light)`
$\top$	$\{red : 255, green : 0, blue : 0\}$
$\perp$	$\perp$

The main advantage of LIDL interaction expressions is that they are very general. Many first-class constructions of other programming languages can be represented as LIDL interactions. As an example, the following table quickly summarises the semantics of the `()=()` interaction. Note that this interaction complies with the **`in`** interface, indeed, its behaviour consists in sending to the left-hand-side the value it receives on the right-hand-side, *only* when the interaction is active.

`((x)=(y))`	`(y)`	`(x)`
⊤	5	5
⊤	⊥	⊥
⊥	5	⊥
⊥	⊥	⊥

LIDL programs structure

LIDL programs structure is similar to functional programs structure. Functional programs are represented as a function. A LIDL program is nothing more than an interaction.

The same way that functional programming languages use function signatures to define functions, LIDL use interaction signatures. Since LIDL uses interfaces instead of data types, interaction signatures are described in terms of interfaces.

As an example, here is the signature of the interaction `when()then()` which is instantiated as the root of the example interaction expression of Listing 2:

```
1 ( when (condition: Boolean in)
2   then (effect: Activation out)
3 ): Activation in
```

Listing 3. The signature of an interaction

The same way that functional programming languages allow to define functions by specifying a signature and the expression it reduces to, LIDL allow to define interactions by specifying a signature and the expression it reduces to.

As an example, here is the definition of the interaction `turn()red` which is used in our example expression of Figure 4:

```
1 interaction
2   (turn (thing: Color out) red): Activation in
3 is
4   ((thing)=({red:(255),green:(0),blue:(0)}))
```

Listing 4. Complete LIDL definition of an interaction

Finally, the same way a functional programmer composes functions in order to make more complex functions, a LIDL programmer composes simple interactions in order to make more complex interactions, ending with a final complex interaction: the LIDL program itself.

USE OF LIDL PROGRAMS

LIDL is only a convenient textual way to describe Directed Acyclic Graph (DAG) structures. Indeed, the compiler first expands interactions into base interactions, using definitions. Then it assigns data flow directions using interfaces definitions. This results in a DAG which express the transition function of a state machine. As an example, Figure 5 shows the graph associated with our example expression.

It is really important to notice that the graph shown in Figure 5 is really nothing more than a graph ordering of the graph shown in Figure 4, with data dependency as the ordering relationship. Data dependency is easily inferred from the interfaces.

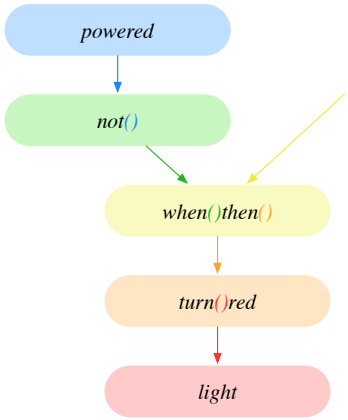

Figure 5. The example expression compiled into a directed acyclic graph

This graph representation is in fact Single Static Assignment (SSA) form [9] of the executable implementing the specified interaction. This form allows different uses such as optimisations, verification, proofs and code generation.

Optimisation

Optimisation can be performed by analysing the graph representation, and generating different execution schemes depending on the requested inputs and outputs, using techniques such as push (data driven evaluation) and pull (demand driven evaluation) as applied to functional reactive programming in [15].

Verification

Verification and proof can be performed by transforming intermediate representation into state machines. The graph representation exactly describes the transition function of such a system, while the state vector is easily derived. It is important to note that the only way for data to persist from one execution step to the next is to be part of a `previous()` interaction. Hence, the state vector is *exactly* the set of interactions which are included in `previous()` interactions. Finally, the generated system has a structure which is very similar to systems generated by other synchronous data flow programming languages such as Scade or Lustre [17]. This potentially allows to leverage the verification tools that have been developed and used for these languages.

Code generation

Code generation has two main objectives: prototype code generation, and production code generation. Both are similar in nature, and are made relatively easy thanks to the intermediate representation. The target languages only have to provide a few features: compound data types, functions, and data types corresponding to LIDL basic data types. At the moment, code generation tools are being developed for two languages: The first is Javascript, in order to enable quick

prototyping in a web app, and even some sort of Read-Eval-Print-Loop similar to the one available online for the Elm functional reactive programming language [10]. The other target language is C, as it is probably the most common language in use for critical systems.

Human models and automatic testing

We have seen that LIDL can be used to specify interactions to be performed by computers. It is also a surprisingly convenient way to model interactions to be performed by human agents.

The high abstraction capabilities of LIDL coupled with its close-to-natural-language syntax allows to specify human interactions associated with a system in a very formal way, while remaining similar to a user manual. LIDL descriptions of human interactions are interesting because they bridge the gap between *task models* and *user manuals* [8], being a generalisation of both.

Typically, LIDL developers would code two things: computer-side interactions (e.g a widget behaviour), and their human-side counterparts (e.g. how to use a widget). This approach is similar to test driven development, applied to interactive systems. This formal specification of human interactions enables automatic testing, by executing a system composed of the computer-side interactions on one side, and the human-side interaction on the other, in an approach similar to [5].

Furthermore, LIDL makes it easy to take into account and model human errors and non deterministic behaviour such as those detailed in [7] and [19]. This allows to test interactive systems even more completely, by simulating the consequences of human errors. Listing 5 shows an example human-side interaction that details how to click on a button, taking into account one error type: omission. The `either()()` interaction represents a non-deterministic choice.

```
interaction
  (click on (theButton: Button)): Activation in
is
  ( either
    ((theButton.click)=(active))    // Nominal
    (nothing)                       // Omission !
  )
```

Listing 5. LIDL definition of a potentially faulty human interaction

USE CASE

In this section, we will use LIDL to describe the Boiling Water Reactor (BWR) use case. For the sake of simplicity, we will limit ourselves to an abstract interface as described in [22]. However, LIDL is not restricted to the specification of abstract user interfaces.

LIDL puts an emphasis on reusability. In the use case, this means that we will take advantage of the similarities between components in order to limit the bulk of code. Figure 6 shows common elements in coloured frames, these common elements will be coded as reusable components.

LIDL implementation of a basic component

Figure 6. A screenshot of the BWR simulator with some common elements outlined in common colors

To get started, let's look at the implementation of a simple abstract slider, which could be part of a standard abstract widget library for LIDL.

Listing 6 shows the interface that this abstract slider complies to. The abstract slider outputs two things to the user: The value of the slider, concretely implemented by the position of the cursor, and the slider range, concretely implemented by the labels at each end of the slider. The abstract slider has one input from the user: The position that the user wants the slider to be at.

```
interface Slider is
  {
    value: Number out,
    range: {min: Number, max: Number} out,
    selection: Number in
  }
```

Listing 6. The interface of an abstract slider

We could define many interactions that implement this interface. Listing 7 presents one of them. This implementation follows these arbitrary design choices:

- It takes an `enabled` argument that specifies if the slider is enabled or not.
- It takes two arguments to specify the range of the slider.
- In case no value is provided for the slider position, it will initialise as the lower bound of the range.
- It sets the value of the argument `theSelection` when changed by the user, or when the range is changed so that it becomes incompatible with the previous value of the slider.
- It take an argument `constrainedPosition` that allows to programatically set the value of the slider, overriding user input.

Several interactions are used in order to define the slider interaction. For example, note the use of the `()fallbackto()fallbackto...` interaction (lines 16-19). This interaction uses the activation of its arguments, and picks the first argument which is active.

Another important point to notice is the argument named `theSelection` (line 5). Since it is an **out**, it will not be read in order to compute a result. In fact, it will be written to, i.e. a value will be sent to it. This is unlike other arguments that are **in**, which have roles similar to arguments programmers are used to.

By looking at this implementation of the slider, it is really easy to notice that it is a stateful component. Indeed we can see a `previous()` interaction (line 18). The interaction inside the `previous()` is `currentValue`, so `currentValue` is the state variable.

```
1  interaction
2    ( slider (enabled: Activation in)
3      between (min:Number in) and (max:Number in)
4      constrained to (constrainedPosition: Number in)
5      selecting (theSelection: Number out)
6    ): Slider
7  is
8    ((when (enabled)
9      then ({
10       value:(currentValue),
11       range:({min:(min),max:(max)}),
12       selection:(userInput)
13     }))
14    behaviour
15     ((current value)=
16       (((constrainedPosition)
17         fallback to (userInput)
18         fallback to (previous(currentValue))
19         fallback to (min))
20       kept between (min) and (max))
21     ))
22 with
23   interaction (currentValue):Number ref
24   interaction (userInput):Number ref
```

Listing 7. The definition of an abstract slider interaction

Listing 8 shows an example use of the slider defined in Listing 7. This instance will always be enabled, because the `enabled` argument is set to the constant `active`. Since the constrained value is set to `inactive`, this instance will allow the user to select a number in the constant range $[0, 2000]$, and the value selected by the user will be sent to a variable named `myValue`.

```
1 (slider (active) between (0) and (2000)
2   constrained to (inactive) selecting (myValue))
```

Listing 8. An instance of the abstract slider

LIDL implementation of a compound component

We will describe the components framed in green on Figure 6. These components, that we will call "complex sliders", are composed of:

- A label indicating the purpose of the slider to the user
- A slider allowing the user to select a value. The slider is the one defined in the previous section
- A toggle button to switch between manual and auto modes
- A label indicating the value and units of the selection

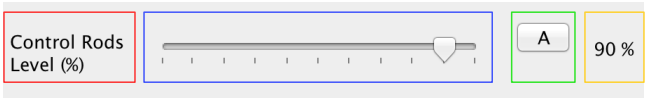

Figure 7. A screenshot of a complex slider component

Figure 7 shows the concrete implementation of this complex slider, and Listing 9 shows its LIDL interface. Note that it reuses the `Slider` interface defined in the previous section, as well as other interfaces.

```
1 interface ComplexSlider is
2   {
3     title: Label,
4     slider: Slider,
5     toggle: ToggleButton,
6     value: Label
7   }
```

Listing 9. The interface of the complex slider

Listing 10 shows an implementation of this complex slider.

```
1  interaction (
2    complex slider
3    named (title: Text in)
4    between (min:Number in) and (max:Number in)
5    (units:Text in)
6    constrained to (constrainedPosition: Number in)
7    selecting (theSelection: Number out)
8    requesting (mode: Activation out) automation
9    ):ComplexSlider
10 is
11   ({
12     title: (
13       label (active)
14       displaying (title)),
15     slider: (
16       slider (active)
17       between (min) and (max)
18       constrained to (constrainedPosition)
19       selecting (theSelection)),
20     toggle: (
21       toggle (active)
22       pushed (when(constrainedPosition))
23       displaying ("A")
24       toggling (mode)),
25     value: (
26       label (active)
27       displaying ((theSelection) " " (units)) )
28   })
```

Listing 10. Definition of a complex slider interaction

Listing 11 shows an example instance of this complex slider, corresponding to the concrete implementation depicted in Figure 7.

```
1 ( complex slider
2   named ("Control Rods Level")
3   between (0) and (100) ("%")
4   constrained to (controlRodsAutoValue)
5   selecting (controlRodsLevel)
6   requesting (controlRodsAutoMode) automation
7 )
```

Listing 11. An instance of the complex slider interaction

CONCLUSION

This paper presented a quick overview of LIDL, a language dedicated to the description of interactions, and a use case. The use case showed that LIDL allows to specify safe complex behaviour. In particular, it is noteworthy that, as compared to other approaches, the LIDL way of thinking as two consequences:

- Removing duplicate or boilerplate code as seen in other languages, such as getter/setters and observer pattern functions. This is noticeable by the relatively small size of LIDL programs.
- Forcing designers into thinking about the actual interaction, enforcing to explicitly define aspects that are usually implicit or merged into objects whose semantics are not clear. This is noticeable in the slider example (Listing 7), where an explicit distinction is made between the user input and the slider current value. This explicit distinction allows to have a sane behaviour, even when the slider range is dynamically changed, while keeping the code simple.

LIDL is only a language. Architectural concepts that fit with LIDL are being developed, but not detailled in this paper. The architectural ideas behind LIDL converge with those recently presented in [16] and similar approaches around unidirectional data flow. A general framework for the specification of abstraction levels of interactive systems inspired by [24] is being developed in parallel with LIDL.

REFERENCES

1. 2012. DO178C. Software Consideration in Airborn Systems and Equipment Certification, release C. (2012). RTCA,Inc.
2. Gregory D Abowd and Alan J Dix. 1994. Integrating status and event phenomena in formal specifications of interactive systems. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering* 22 (December 1994), 23.
3. G. A. Agha. 1986. Actors: a model of concurrent computations in distributed systems. *MIT Press* (1986).
4. ARINC 2012. *Specification 661* (supplement 5, draft 1 ed.). ARINC. http://www.aviation-ia.com/aeec/projects/cds/index.html
5. Eric Barboni, Jean-François Ladry, David Navarre, Philippe Palanque, and Marco Winckler. 2010. Beyond Modelling: An Integrated Environment Supporting Co-execution of Tasks and Systems Models. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*. ACM, New York, NY, USA, 165–174. DOI: http://dx.doi.org/10.1145/1822018.1822043
6. BEA. 2012. *Rapport final sur l'accident survenu le 1er juin 2009 à l'Airbus A330-203 immatriculé F-GZCP exploité par Air France, vol AF 447 Rio de Janeiro - Paris*. Technical Report. Direction Générale de l'Aviation Civile. http://www.bea.aero/docspa/2009/f-cp090601/pdf/f-cp090601.pdf.
7. M.L. Bolton and E.J. Bass. 2011. Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*. 1788 –1794. DOI: http://dx.doi.org/10.1109/ICSMC.2011.6083931
8. Judy Bowen and Steve Reeves. 2012. Modelling User Manuals of Modal Medical Devices and Learning from the Experience. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '12)*. ACM, New York, NY, USA, 121–130. DOI: http://dx.doi.org/10.1145/2305484.2305505
9. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct 1991), 451–490. http://doi.acm.org/10.1145/115372.115320
10. Evan Czaplicki. 2012. Elm: Concurrent FRP for Functional GUIs. (2012).
11. A. Dix and C. Runciman. 1985. Abstract models of interactive systems. In *Proceedings of the HCI'85 Conference on People and Computers: Designing the Interface*. 13–22.
12. Alan John Dix. 1991. *Formal methods for interactive systems*. Academic Press.
13. D.J. Duke and M.D. Harrison. 1993. Abstract Interaction Objects. *Computer Graphics Forum* 12, 3 (1993), 25–36. DOI: http://dx.doi.org/10.1111/1467-8659.1230025
14. ECMA. 2013. The JSON Data Interchange Format. (October 2013). http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
15. Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 25–36. DOI: http://dx.doi.org/10.1145/1596638.1596643
16. Facebook. 2013. React - a JavaScript library for building user interfaces. (2013). http://facebook.github.io/react/
17. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous dataflow programming language Lustre. In *Proceedings of IEEE (79)*. 1305–1320.
18. D. Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274.
19. Célia Martinie and Philipe Palanque. 2014. Fine Grain Modeling of Task Deviations for Assessing Qualitatively

the Impact of Both System Failures and Human Error on Operator Performance. In *2014 AAAI Spring Symposium Series*.

20. David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. 2009. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.* 16, Article 18 (November 2009), 56 pages. Issue 4. DOI: http://dx.doi.org/10.1145/1614390.1614393

21. F. Paternò and G. Faconti. 1992. On the use of LOTOS to describe graphical interaction. In *Proceedings of the HCI'92 Conference on People and Computers*. 155–173.

22. Fabio Paterno, Carmen Santoro, and Lucio Davide Spano. 2009. MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments. *ACM Trans. Comput.-Hum. Interact.* 16, 4, Article 19 (Nov. 2009), 30 pages. DOI: http://dx.doi.org/10.1145/1614390.1614394

23. Esterel Technologies. 2011. Scade Display. (2011). http://www.esterel-technologies.com/products/scade-display

24. Pamela Zave and Jennifer Rexford. 2012. The Geomorphic View of Networking: A Network Model and Its Uses. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing (MW4NG '12)*. ACM, New York, NY, USA, Article 1, 6 pages. DOI: http://dx.doi.org/10.1145/2405178.2405179

Layers, resources and property templates in the specification and analysis of two interactive systems

José Creissac Campos
Dep. Informática /
Universidade do Minho &
HASLab / INESC TEC
Braga, Portugal
jose.campos@di.uminho.pt

Paul Curzon and Paolo Masci
EECS, Queen Mary University of London
Mile End, London E1 4NS, UK
p.curzon@qmul.ac.uk,
p.m.masci@qmul.ac.uk

Michael Harrison
School of Computing Science,
Newcastle University,
Newcastle-upon-Tyne, UK
Universidade do Minho &
HASLab/INESC TEC, Queen Mary University London
michael.harrison@ncl.ac.uk

ABSTRACT

The paper briefly explores a layered approach to the analysis of two interactive systems (Nuclear Control and Air Traffic Control), indicating how the analysis enables exploration of the particular features emphasised by the use cases relating to the examples. These features relate to the interactive behaviour of the systems. To facilitate the analysis, property templates are proposed as heuristics for developing appropriate requirements for the respective user interfaces.

Author Keywords

Formal methods, interactive systems, usability heuristics

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

Formal modelling can have substantial benefits when they enable clarification of the assumptions made about a design. This paper looks at two case studies, provided as part of the preparation for the Workshop on Formal Methods in Human Computer Interaction. In the two cases to be discussed, formalism has already been used by the providers of the case studies to explore different features of the two examples. The nature of their analyses and the illustrative examples suggest different approaches to analysis in the two cases. The first case appears to focus on features of the device interface while the second case uses a notation for describing the tasks supported by the interface. In this paper we indicate how a common modelling approach based on layers can be used to specify the systems, enabling clear distinctions between levels of analysis while at the same time maintaining the integrity of the specification. We further comment that the analysis of properties in relation to these layers can be facilitated through the use of property templates.

To cite this article use the following information:

José Creissac Campos, Paul Curzon, Paolo Masci, and Michael Harrison. 2015. Layers, resources and property templates in the specification and analysis of two interactive systems. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 38-43. http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

THE USE CASES

Two examples of safety critical interactive systems[1] discussed in the paper illustrate two distinct and important focuses commonly found in the analysis of interactive systems. Both examples present the interactive behaviour of the system, and a description of normative tasks that should be followed by operators to use the system. The first, the nuclear control example, focuses on the user interface and on checklists[2], while the second, the air traffic control (ATC) example, focuses on the user interface and distributed tasks carried out by operators and pilots in coordination.

In the first case the analysis is concerned with the role of an operator in interacting with a device. It is concerned with whether the operator can control aspects of the system in a clearly understandable way and be aware of the situation and the recovery mechanism if a failure occurs that can only be managed by the protection system. The focus of a model of the nuclear use case would be the display, the graphics, the status display, the sliders, the enabled actions and how these change the display. It would also be concerned with how effectively the protection system supports failure events. The properties of the protection system will be concerned with whether the system blocks the user effectively when unsafe actions are detected, and whether the user can trace the process through the information provided by the interface.

In the second use case there are more details about the tasks that controllers and the pilots are engaged in. The two operators have different roles and these roles are made explicit through a notation for describing the tasks. The question invited by the ATC description is how the arrival sequence display supports their activities and roles.

Whatever the level of analysis of the user interface, there are low level questions about the underlying system that are necessary to understand the design of the user interface. These

[1] `https://sites.google.com/site/wsfomchi/use-cases` (downloaded 28/5/15)

[2] `http://www.hci-modeling.org/nppsimulator/BWRSimulationDescription.pdf` (downloaded 28/5/15)

include: understanding which parts of the underlying process are visible in the user interfaces; which user actions produce visible feedback that can help the operators assess what has been done, and what has been achieved; and whether there are modes, and how transparent the effect of these modes is. In fact, interactive systems of any complexity have a common characteristic that some elements of the state of the system are perceivable (for example, visible or audible), and that user actions transform the state [5]. Furthermore, not all actions are permitted all of the time, and the behaviour of actions can depend on distinguished state attributes called *modes*, see [6] for further discussion.

STRUCTURE OF THE MODELS

It is important to distinguish between interactive systems and the components of the interactive systems. Interactive systems are socio-technical systems involving people, devices, and artefacts (desks, pieces of paper, pens, tablets and so on). The primary focus of the modelling approach illustrated here is based on the models of the interactive devices that are a part of the interactive system, whilst the property templates presented capture aspects of the system that can facilitate device-user interaction.

Users have difficulty understanding the progress of a system when elements of the state of the system, that are relevant to that understanding, are not visible in a form that makes sense to them. At the same time, confusion can arise when actions relevant to the current activity are either apparently or actually disabled by the system, or when the actions have an unexpected or inconsistent effect with respect to the users' knowledge and experiences of the system. Actions and states are therefore elemental in understanding interactive behaviour. Modes are also important. It is unusual that an interactive system is so simple that actions always have the same effect.

To achieve the goals and activities required of the users, most interactive systems are designed more or less effectively to ensure that the information required (we call them *information resources* [2]) are made explicitly available, and in a form that can be easily understood by the users. A role of a model of the interactive system is therefore to make these information resources explicit so that assumptions about the constraints they impose may be analysed.

The interface specification

Interactive systems can be specified by defining the set of actions, including user actions, that are possible within them. These actions affect the state of the system and are affected by many attributes of the state of the system. In the case of the interactive device they are often determined by the mode of the device. The model of the interactive system we develop aims to make explicit the relevant information resources needed for the analysis of the interactive behaviour of the system and that includes models of the interactive devices as well as the particular actions that define the activities that are the work of the system. The interface specification describes what the display shows and captures the effects of user level actions. The display will show some features of the state of the reactor, these features may be encoded as part of the interface. It will also show the user actions that are translated into actions within the reactor. The specification includes display widgets: showing simple status information. These include the RKS, RKT, KNT, TBN, $WP1$, $WP2$,CP, AU. These displays are associated with a range of colours. The change of colour presumably results from some combination of reactor states not made clear in the documentation. The display also shows actions associated with the valves: $SV1$, $SV2$,$WV1$, $WV2$.

Analysis of an interactive device is then concerned with proving that relevant feedback is given on completing an action, that relevant information is available before an action is carried out, that it is possible to recover from an action in specified circumstances, that it is always possible simply to step to some home mode whatever the state of the device and that actions can be completed consistently.

Structuring specifications

We structure our model of the interactive system as four layers. The first layer simply specifies the constants and types used throughout the specification. It includes types relating to the devices involved and the entities that are in the broader system. For example, in the case of the reactor these types would include pressure and volume defined as part of an aggregate type defining the tanks. There would also be types associated with pumps and valves. Constants would include maximum and minimum values that could raise error events in certain situations.

The second layer describes assumptions about the underlying process, managed or controlled by the devices, that are required as a basis for understanding the user interface specification. In practice this layer is often reused across families of device models when exploring the effects of differing user interfaces [7]. It will describe the most primitive representation of the nuclear process or the aircraft space required to consider the interactive system. A specification of the underlying reactor, describing the details of the relation between reactor core and turbine, would include attributes defining water level and pressure for each. The specification at this level would also define the characteristics of the pumps and valves. The pumps would be associated with rates per minute and the valves would be on or off. A number of actions will be specified at this level. An action $tick$ would represent the interval of one minute and update the levels and pressure depending on the setting of pump and valve. There will be further actions switching pumps on and off, opening and closing valves and changing the value of flow in the pump.

The third layer describes the interfaces for the various devices used in the interactive system. These models use the process descriptions described in the second layer. They make those aspects of the state visible through the interface explicit. They describe the user actions, including for example how the sliders or buttons or other display widgets work. The third layer of the specification of the nuclear control user interface specifies how the user sets, controls and views the operation of the device. It is specific to this particular interface, whereas the reactor specification may be more generic.

The fourth, and final, layer makes explicit the information resources that are required for different actions in different circumstances. It captures constraints on action based on the goals and activities that the user achieves or carries out [2]. This layer contains an interactive system view. The activities and actions are "resourced" by user interfaces for devices that are used in the interactive system or, indeed, any other source of relevant information that is present within the interactive system. It adds attributes that are not captured by the devices and includes (meta-)actions that describe activities that may involve actions of the interactive devices. An example of this fourth layer used in a different context can be found in [9].

Tool support

Full details of the models that are developed of the two use cases are not the focus of this paper, rather we intend an indication of our approach. Indeed different languages might be used within the context of the approach proposed depending on the type of analysis intended.

Two approaches to specification and proof are possible with the example just given: model checking and theorem proving. In the present case we focus on a theorem proving approach because an important feature of the analysis, that has issues from a user interface point of view, concerns the mechanisms for number entry. Since the domain of numbers is relatively large, proof using model checking can result in analyses of very large models that can be intractable.

The automated theorem prover used is *Prototype Verification System (PVS)* [11]. It combines an expressive specification language based on higher-order logic with an interactive prover. PVS has been used extensively in several application domains. It is based on higher-order logic with the usual basic types such as `boolean`, `integer` and `real`. New types can be introduced either in a declarative form (these types are called *uninterpreted*), or through *type constructors*. Examples of type constructors that will be used in the paper are function and record types. Function types are denoted `[D -> R]`, where `D` is the domain type and `R` is the range type. Predicates are Boolean-valued functions. Record types are defined by listing the field names and their types between square brackets and hash symbols. Predicate subtyping is a language mechanism used for restricting the domain of a type by using a predicate. An example of a subtype is `{x:A | P(x)}`, which introduces a new type as the subset of those elements of type `A` that satisfy the predicate `P` on `A`. The notation `(P)` is an abbreviation of the subtype expression above. Predicate subtyping is useful for specifying partial functions. Dependent subtypes can be defined, e.g., the range of a function or the type of a field in a record may depend on the value of a function argument or the value of another field in the record, respectively.

Specifications in PVS are expressed as a collection of *theories*, which consist of declarations of names for types and constants, and expressions associated with those names. Theories can be parametrised with types and constants, and can use declarations of other theories by importing them. The prelude is a standard library automatically imported by PVS. It contains a large number of useful definitions and proved facts for types, including among others common base types such as Booleans and numbers (e.g., `nat`, `integer` and `real`), functions, sets, and lists.

The standard format of the specifications is that it contains a definition of a set of actions

```
action: TYPE = [state -> state]
```

which are permitted in particular situations, sometimes all situations. For each action there is a predicate

```
per_action: TYPE = [state -> boolean]
```

that indicates whether the action is permitted.

MODELLING THE CASE STUDIES

Model of the Nuclear Control User Interface

> "The operation of a nuclear power plant includes the full manual or partially manual starting and shut down of the reactor, adjusting the produced amount of electrical energy, changing the degree of automation by activating or deactivating the automated steering of certain elements of the plant, and the handling of exceptional circumstances. In case of the latter, the reactor operator primarily observes the process because the safety system of today's reactors suspends the operator step by step from the control of the reactor to return the system back to a safe state."

The interface involves schematics of the process, the availability of actions as buttons and graphical indications of key parameters, for example temperature and levels. The specification of the model can be layered according to the levels described above as follows. **The first layer** includes definitions of constants such as the maximum and minimum water levels in the reactor tank and condenser.

```
min_wl: nonneg_real
max_wl: {x: nonneg_real | x>min_wl}
```

The second layer specifies those aspects of the underlying reactor that are required to produce a model of the interface. It describes details of the relation between reactor core and turbine. These details include state attributes defining water level and pressure for each component of the core and turbine. It also includes a definition of the characteristics of the pumps and valves. Pump behaviour is abstracted as a number representing the pumping rate. Valves are abstracted as on/off switches. Actions are specified that model the events that are automatically triggered within the system. For example, an action $tick$ represents the periodic update of rate and pressure depending on the setting of pump and valve.

```
tick(st: process_state): process_state =
  p WITH [ reactor := tick(p`reactor)
           condensor := tick(p`condensor)
         ]
...
tick(r: reactor_state): reactor_state =
  r WITH [ rate := ...,
           pressure :=  ...
         ]
```

Further actions represent functions for switching pumps on and off, opening and closing valves and changing the value of flow in the pump.

The third layer describes what the reactor user interface shows on displays, what user actions are permitted, and how the system changes state in response to user actions. For the considered user interface, the model includes a specification of the actual status indicators (RKS, RKT, etc.), as well as the level and pressure of reactor and condenser.

```
state: TYPE = [#
    r: process_state
    rksm, rkt: Colour,
    rct_pressure: nonneg_real,
    SV1_state_open: boolean,
    ... #]
```

The colours of the indicators are linked to states of the underlying reactor (modelled in the second layer). The model also specifies that the user can perform open/close actions on valves, change the level of control rods, and change the rate of the pumps by interacting with controls on the user interface. All these actions are defined in terms of actions specified in the second layer of the model.

```
click_close_SV1(st: state): state =
   COND st'SV1_state_open ->  st WITH
    [r :=
     close_valve(st`process_state,sv1)],
    ELSE -> st
   ENDCOND
```

The fourth layer describes constraints on the action offered by the user interface based on the goals and activities that the user achieves or carries out [2]. For example, the action **OpenSV1** which opens a particular valve in the reactor will be appropriate in certain circumstances and for particular purposes. The information required by the operator to judge those circumstances should be visible to the operator. This information includes water levels and pressures for the relevant tank. To enable specification of these constraints an understanding of the supported activities is required. The effect of this layer of specification is to further constrain the behaviours of the user interface model to intended or plausible behaviours. The purpose of this constraint is to consider whether plausible behaviours are excluded or whether additional behaviours would be allowed by the specification that could indicate user confusions.

Actions may be specified at the level of user activity in this layer. For example, consider the user activity $recover$ in contrast to the autonomous action that causes recovery. This action would specify constraints. For example, it would specify that "increasing pressure" using the relevant action in the third layer would occur only if other actions had already been completed and the displayed tank, valve and pump parameters specified in the second layer were displayed (in the third layer) indicating particular values.

Further activities include for example "monitor recovery". This would be expressed as an action that describes the constraints on the operator when monitoring an autonomous recovery. The specification of the action would include the information resources that would be required in the monitoring process at different stages of the recovery and would specify the conditions in which any user actions would take place.

Model of the Air Traffic Controller Radar Screen

> "The AMAN (Arrival MANager) tool is a software planning tool suggesting to the air traffic controller an arrival sequence of aircraft and providing support in establishing the optimal aircraft approach routes. Its main aims are to assist the controller to optimize the runway capacity (sequence) and/or to regulate/manage (meter) the flow of aircraft entering the airspace ... '

In this case **the first layer** defines constants such as known constraints for flight (e.g., aircraft performance model parameters and constraints) and runways (e.g., maximum capacity).

The second layer captures the logic of the arrival manager planning software for suggesting arrival sequence and optimal approach routes. State attributes would specify dynamic parameters of the system, like flight plan, radar data, and weather information. An action $tick$ specifies how the suggested arrival sequence and optimal approaches are updated by the system on the basis of the actual values of flight plans, radar data, etc. Further actions specify the logic for updating dynamic parameters of the system. These will include: a trajectory predictor algorithm, a sequencer module, a weather data source, etc. Additional actions can be introduced for modelling more complex scenarios in which pilots can request emergency landing.

It is worth noting that each action is a self-contained description of how the system state changes when a given event occurs. Because of this, although adding new actions to the model makes the overall behaviour of the model more complex, it does not necessarily increase the complexity of the model. The same applies for the complexity of the analysis: if a new action does not affect the value of state variables relevant to the analysis of a property, then the complexity of the analysis of that property remains unchanged with and without the new action. This is true of theorem proving. With model checking this analysis is less straightforward.

The third layer describes what information is presented on the screens to the plan controller and executive controller. The model of the arrival manager display will therefore include a specification of the arrival timeline, time-management information, aircraft callsign, and wake turbulence category. The model of the radar screen display will include which aircraft labels are visible, their positions on the screen, and their speed vectors.

The fourth layer is based on the task descriptions provided in the use case. The task representations provide the display context required to constrain the actions. In this case actions will be activity actions, actions that are permitted by the states of the display but do not themselves change the display. They specify the assumptions about when the operators' actions are permitted.

Issues

Models of the type outlined have been developed for other interactive systems using both a model checking approach and a theorem proving approach [9, 7, 8, 1]. The advantage of model checking is that it is possible to explore, more readily, reachability properties as well as potential non-determinisms. The disadvantage is that the size of model is seriously limited. While it is possible to explore the essential details of the control of the nuclear plant using a model checking approach, this is not possible of the ATC system. Aspects such as the trajectory predictor algorithm mean the second layer of the model would be too complex. Making it abstract enough to make analysis feasible, would restrict what could be asked of the model, in terms of relevant properties to prove, making the analysis less relevant.

Theorem proving allows analysis of larger models but properties may be more difficult to formulate and prove. In particular, while model checking allows simple formulations of reachability properties, these are difficult to specify using a theorem proving approach.

There is a tradeoff to be made between the effort needed to develop a model amenable for verification and the effort need to carry out the proofs. Typically a theorem proving based approach will gain advantage in the former, because of more expressive languages, and model checking in the latter, because of more automated analysis. In all cases, how to identify and express the properties of interest is also an issue.

PROPERTY TEMPLATES

The analysis approach uses property templates as heuristics to generate properties that are tailored to the device. Tailoring the heuristics leads to insight about the device design as well as producing properties that will, if true of the design, lead to an interface being more predictable and easy to use. The heuristics that will be considered in more detail and were described in [3] are: *completeness*, *consistency*, *feedback*, *reversibility* and *visibilty*.

The heuristics have the following characteristics:

completeness captures the notion that it should be possible to reach any other state (more likely mode) in one (or a few) steps. A typical example of this property is that the design has some "home" state and a single action is sufficient to reach that home state regardless of what state the device is currently in. The first use case would suggest a completeness property which ensures that critical actions can always be taken in one step. The user interface never "modes the operator in" so that responding to a critical situation requires a complex interaction.

consistency requires that an action will always change the state of the device in a consistent way. Consistency is also concerned that similar actions have similar effects. These properties can be expressed in a number of ways and require some invention to be assured that a property is of the appropriate form. Examples of consistency properties for the Nuclear Control interface are: control rods can always be stopped; switches for valves on the user interface can always be used to change the valve states. A number of consistency properties are like these based on actions. However there are also properties that specify that a value can only change as a result of a certain type of action, or if the state is in a particular mode. An example is: all steam valves can be closed only when all feed water valves are closed, and the water level in the reactor is stable. An example of a property that related to sets of actions is that all actions that use a slider will involve similar effects.

feedback requires that an action that has an effect on the state of the system has also an effect that can be perceived by the user. For the Nuclear Control interface, an example feedback property will check that the water level indicators of the user interface correctly report changes to the actual value of the corresponding state variable of the reactor. For the Air Traffic Controller interface, an example feedback property will stipulate that all actions that have an effect on the system state will be signposted on the user interface (e.g. whether or not the speed vectors take into account changes in heading is a feedback issue).

reversibility ensures that an action can always be reversed. It is important that certain actions can be reversed. There will be constraints that limit this reversibility. For example it may be the case that an action such as opening a valve can be reversed within a specific time interval only in certain circumstances to prevent an inadvertent and extremely costly action.

visibility specifies that a state attribute in the second layer of the model is always "mirrored" by an appropriate state attribute in the third layer model. An example of such a property is that the tank level is always represented by the tank graphic in the interface or that the ATM display always correctly represents the states of the aircraft that are on approach.

For each heuristics a template is provided to express properties relevant for the heuristics. In its more general form these properties are expressed over a state machine. Hence, in its simplest form, the fact that action *ac* causes a perceivable effect (captuted by $effect_{percv}$) is expressed by

$$\forall_{s \in State} \bullet effect_{percv}(s, ac(s))$$

For the model checking case, CTL (Computational Tree Logic) and LTL (Linar Time Logic) (see [4] for an introduction) templates are provided by the IVY tool [7]. For PVS, translation of the templates is relatively straightforward, resorting to induction over the reachable states of the model.

The instances of the properties generated from the templates are usually described in terms of concepts of the third layer model. The significance of the fourth layer is that it constrains the paths for which the properties are true. The fourth layer of the specification identifies sets of plausible behaviours and in many cases the properties to be considered are required to be true only for these behaviours. For example it may be appropriate to require that any action that may occur in a path constrained by information resources will have visible feedback.

DISCUSSION AND CONCLUSIONS

Two approaches to specification and proof are possible with the considered examples: model checking and theorem proving. Model checking is the more intuitive of the two approaches. The language adopted Modal Action Logic with interactors (MAL) [3] expresses state transition behaviour in a way that is more acceptable to non-experts. The problem with model checking is that state explosion can compromise the tractability of the model so that properties to be proved are not feasible. Model checking, hence, is more convenient for analysing high level behaviour, for example when checking the modal behaviour of the user interface. Theorem proving, while being more complex to apply, provides more expressive power. This makes it more suitable when verifying properties requiring a high level of details, such as those related to a number entry system, because the domain of numbers is relatively large.

To employ the strengths of the two approaches simple rules have been used to translate from the MAL model to the PVS model that is used for theorem proving. Actions are modelled as state transformations, and permissions that are used in MAL to specify when an action is permitted are described as predicates. The details of the specification carefully reflects its MAL equivalent. This enables us to move between the notations and verification tools, choosing the more appropriate tool for the verification goals at hand.

One aspect that has not been discussed herein is the analysis and interpretation of verification results. The possibility of animating the formal models to create prototypes of the modelled interfaces, and the possibilities these prototypes raise in terms of discussing the results of verification with stakeholders has been discussed in [10]. Such prototypes can be used either to *replay* traces produced by a model checker or interactively to both discuss the findings of the verification or help identify relevant features of the system that should be addressed by formal analysis.

ACKNOWLEDGMENTS

José Creissac Campos and Michael Harrison were funded by project ref. NORTE-07-0124-FEDER-000062, co-financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese foundation for science and technology (FCT). Paul Curzon, Michael Harrison and Paolo Masci were funded by the CHI+MED project: Multidisciplinary Computer Human Interaction Research for the design and safe use of interactive medical devices project, UK EPSRC Grant Number EP/G059063/1.

REFERENCES

1. Campos, J., Sousa, M., Alves, M. C. B., and Harrison, M. Formal verification of a space system's user interface with the ivy workbench. *IEEE Transactions of Human Machine Systems* (2015). In Press.
2. Campos, J. C., Doherty, G., and Harrison, M. D. Analysing interactive devices based on information resource constraints. *International Journal of Human-Computer Studies 72* (2014), 284–297.
3. Campos, J. C., and Harrison, M. D. Systematic analysis of control panel interfaces using formal tools. In *Interactive systems: Design, Specification and Verification, DSVIS '08*, N. Graham and P. Palanque, Eds., no. 5136 in Springer Lecture Notes in Computer Science, Springer-Verlag (2008), 72–85.
4. Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. MIT Press, 1999.
5. Duke, D. J., and Harrison, M. D. Abstract interaction objects. *Computer Graphics Forum 12*, 3 (1993), 25–36.
6. Gow, J., Thimbleby, H., and Cairns, P. Automatic critiques of interface modes. In *Proceedings 12th International Workshop on the Design, Specification and Verification of Interactive Systems*, S. Gilroy and M. Harrison, Eds., no. 3941 in Springer Lecture Notes in Computer Science, Springer-Verlag (2006), 201–212.
7. Harrison, M., Campos, J., and Masci, P. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering 11*, 2 (June 2015), 95–111.
8. Harrison, M. D., Masci, P., Campos, J., and Curzon, P. Demonstrating that medical devices satisfy user related safety requirements. In *Proceedings of Fourth Symposium on Foundations of Health Information Engineering and Systems (FHIES) & Sixth Software Engineering in Healthcare (SEHC) Workshop*, Springer-Verlag (2014). accepted.
9. Masci, P., Huang, H., Curzon, P., and Harrison, M. D. Using PVS to investigate incidents through the lens of distributed cognition. In *NASA Formal Methods*, A. Goodloe and S. Person, Eds., vol. 7226 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, 273–278.
10. Masci, P., Oladimeji, P., Curzon, P., and Thimbleby, H. PVSio-web 2.0: Joining PVS to Human-Computer Interaction. In *27th International Conference on Computer Aided Verification (CAV2015)*, Springer (2015). Tool and application examples available at http://www.pvsioweb.org.
11. Shankar, N., Owre, S., Rushby, J. M., and Stringer-Calvert, D. PVS System Guide, PVS Language Reference, PVS Prover Guide, PVS Prelude Library, Abstract Datatypes in PVS, and Theory Interpretations in PVS. Computer Science Laboratory, SRI International, Menlo Park, CA, 1999. Available at `http://pvs.csl.sri.com/documentation.shtml`.

Interactive System Modelling - Combining Models with Different Levels of Abstraction

Judy Bowen
The University of Waikato
Hamilton, New Zealand
jbowen@waikato.ac.nz

Steve Reeves
The University of Waikato
Hamilton, New Zealand
stever@waikato.ac.nz

ABSTRACT

Our approach for modelling interactive systems has been to develop models for the interface and interaction which are light-weight but with an underlying formal semantics. Combined with traditional formal methods to describe functional behaviour this provides the ability to create a single formal model of interactive systems and consider all parts (functionality, user interface and interaction) with the same rigorous level of formality. The ability to convert the different models we use from one notation to another has given us a set of models which describe an interactive system (or parts of that system) at different levels of abstraction. which can be combined into a single model for model-checking, theorem proving *etc*. There are, however, many benefits to using individual models for different purposes throughout the development process.

ACM Classification Keywords

D.2.4. Software Engineering: Software/Program Verification;

Author Keywords

Formal methods, interactive systems, model-checking, safety-critical systems

INTRODUCTION

Developing suitable interfaces for safety-critical systems relies on two things. First, they must be usable in their environments by their users - *i.e*. be developed using a sound user-centred design process and following known HCI principles. Secondly, we must be able to verify and validate the user interface and interaction with the same rigour as the underlying functionality. While we can (we hope) assume the former, the latter is harder, and requires us to develop suitable techniques which not only support these requirements but which will also be useful (and used) by the interface developers of such systems. In the rest of this paper we describe the different models and notations we use for different parts of an interactive system. We also discuss how these can be combined into a single model as well as the benefits provided by using the individual models for specific purposes.

To cite this article use the following information:

Judy Bowen and Steve Reeves. 2015. Interactive System Modelling – Combining Models with Different Levels of Abstraction. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 44-46.
http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

THE MODELS

In the usual way we initially consider interactive systems by way of their two main components (functionality and interface) separately. We rely on a combination of existing languages and models to specify the functionality, and have developed models for the interface and interactive components as described next.

Functional Specification

We typically use the Z specification language [6, 5] for our functional specifications along with the Pro-Z component of the ProB tool for model-checking. However, any similar state-based notation could be substituted. Z gives us the ability to reason about the functional behaviour of the system and do any necessary theorem-proving or model-checking to ensure that the system not only does the right thing (expected behaviour) but also does not do the wrong thing (unexpected behaviour).

Presentation Model

This is a behavioural model of the interface of a system described at the level of interactive components (widgets), their types and behaviours, and where in the interface they appear (or in the case of modal systems, which modes they are enabled in). The presentation model can be derived from early designs of interfaces (such as prototypes, story-boards *etc.*), final implementations, or anything in between. As such they can be produced from the sorts of artefacts interface designers are already working with within a user-centred design process. Each window or dialogue (or mode) is described separately in a component presentation model (*pmodel*) by way of its component widgets which are described using a triple:

```
widget name, widget type, (behaviours)
```

The full interface presentation model is then the concatenation of the *pmodels*, and describes all behaviours of the interface and which widgets provide the behaviours. Behaviours are split into two categories, interactive behaviours (I-behaviours) are those which facilitate navigation through the system (opening and closing new windows *etc.*) or affect only presentational elements of the interface, whereas system behaviours (S-behaviours) provide access to the underlying functionality of the system.

Presentation Interaction Model

The presentation interaction model (PIM) is a state transition diagram where *pmodels* are abstracted into states and transitions are labelled with I-behaviours from those *pmodels*. As such the PIM gives a formal meaning to the I-behaviours as well as providing an abstract transition model of the system's navigational possibilities. The usual 'state explosion' problem associated with using transition systems or finite state automata to model interactive systems is removed by the abstraction of *pmodels* into states, so the size of the model is bounded by the number of individual windows or modes of the system.

Presentation Model Relation

Just as the PIM gives meaning to the I-behaviours of the presentation model, the presentation model relation (PMR) does the same for the S-behaviours. These behaviours represent functional behaviours of the system, which are specified in the formal specification. The PMR is a many-to-one relation from all of the S-behaviours in a presentation model to operations in the specification.

Combining the Models

The models of functionality (specification) and interface (presentation model and PIM) are already coupled via the PMR. However, we can combine the models in other ways which ultimately lead to a single model of the entire system. In addition to the Z specification, we also use the visual language, μCharts [7] which is used to model reactive systems. PIMs can also be represented as μcharts, which provides additional benefits over a simple PIM (including the ability to compose specific sets of behaviours in different charts via a feedback mechanism and embed complex charts into simple states in order to 'hide' complexity) [1]. The μCharts language includes several refinement theories which therefore in turn gives us refinement theories for PIMs (trace refinement is particularly useful as it can be represented as a much more lightweight theory for interfaces based on contractual utility) [2]. The semantics of μCharts is given in Z and there is a direct translation available (via an algorithm and tool) from a μchart to a Z specification [8], this in turn means we have an algorithm and means to turn a PIM into a Z specification [4]. It is this which gives us the ability to create a single model of all parts of an interactive system, which can then be used to prove safety properties about that system for example [3].

Whilst this ability to create a single model to reason about an interactive system in its entirety is beneficial, we have also found that it can be useful to take advantage of the fact that the individual models have different levels of abstraction. Even when we want to investigate the system as a whole, we can focus on specific parts of the system, or specific attributes of its behaviour by using one specific model - or smaller combination of models. We describe this next using the Nuclear Power Plant Case study as an example.

THE NUCLEAR POWER PLANT CASE STUDY

We can describe the functionality given in the description document using a Z specification. So we have a description of the observable states of the system (in a Z schema)

$$
\begin{array}{|l}
\textit{ReactorControl} \\
\hline
sv1 : Valve \\
sv2 : Valve \\
cpUMin : \mathbb{N} \\
wv1 : Valve \\
wv2 : Valve \\
outputMW : \mathbb{N} \\
waterlevelMM : \mathbb{N} \\
wp1UMin : \mathbb{N} \\
rodposition : \mathbb{N} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{Stable} \\
\hline
\Xi ReactorControl \\
\hline
cpUMin = 1600 \\
waterlevelMM = 2100 \\
outputMW = 700 \\
\hline
\end{array}
$$

$$Status \mathrel{\widehat=} Stable \lor Error1 \lor SCRAM$$

Figure 1. Part of Z Specification

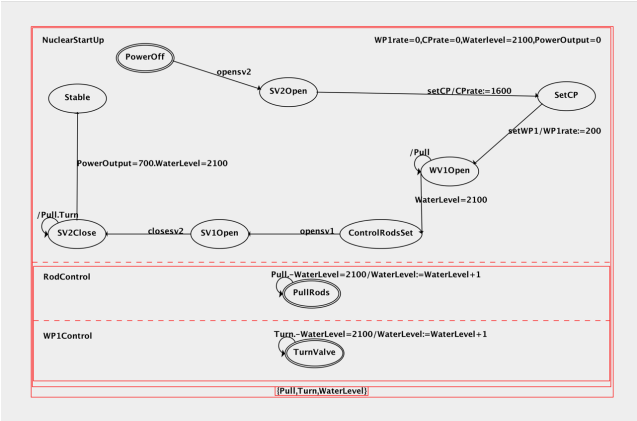

Figure 2. Startup procedure μchart

along with the operations that can change the state of that system. Using a model-checker, such as Pro-B we can then investigate the system to ensure it behaves as expected, for example we can give a description of the system in its stable state and show that when this is not true, the system will be in one of the two error control states - abnormal operation or SCRAM. We could also investigate the described 'Start-up' and 'Shut-Down' steps in a similar manner using the Z, however in order to more easily consider the user inputs to control these procedures we can instead create a μchart which shows the required input levels and reactions that occur in these processes. In figure 2 we show a μchart describing the 'Start-up' procedure. The transitions between the various states the system goes through are guarded by required values on key indicators (such as water level, power output *etc.*) as well as user operations (such as opening and closing valves). In this model we do not distinguish between user operations, system controlled operations and functional monitoring of values. We are most interested here in ensuring the correct outcomes are reached depending on the values and that the components interact properly as shown by the feedback mechanism of the composed charts. Using the Z semantics of μCharts we can then model-check this component of

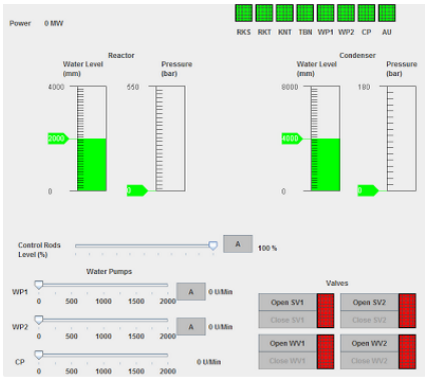

Figure 3. Simulation interface

behaviour or use theorem-proving to ensure that the system progresses correctly through the start-up procedure (and similary shut-down) only when the correct pre/post conditions are met.

Once we are satisfied that the system will behave correctly as described we also need to ensure that the users can perform the required operations and that at the very least the interface provides the necessary controls (we do not talk about the issue of usability of the interface in this paper, however it is of course equally important in ensuring the system can be used). In the nuclear power plant control system we may start with an initial design, such as that given by the provided simulator and ensure that it does indeed provide all of the required behaviour. From the simulation interface shown in figure 3 we can derive a presentation model and PMR (a snippet of these is shown below).

Simulator is
PowerDisplay, Display, (S_OutputPower),
RWaterLevelDisplay, Display,
    (S_OutputReactorWaterLevel),
RPressureDisplay, Display, (S_OutputReactorPressure),
ControlRodCtrl, ActionControl (S_RaiseControlRods,
    S_LowerControlRods),
WP1Ctrl, ActionControl(S_IncWaterPressure1,
    S_DecWaterPressure1),
CPCtrl, ActionControl(S_IncCPressure, S_DecCPressure),
SV1Open, ActionControl(S_OpenSV1),
SV1Close, ActionControl(S_CloseSV1)

S_RaiseControlRods $\mapsto$ RaiseRods
S_LowerControlRods $\mapsto$ LowerRods
S_IncWaterPressure1 $\mapsto$ IncreaseWaterPressure
S_DecWaterPressure1 $\mapsto$ DecreaseWaterPressure
S_OpenSV1 $\mapsto$ OpenSV1
S_CloseSV1 $\mapsto$ CloseSV1

This ensures that all of the required operations are supported by the user interface. We can then use these models to help derive alternate (restricted) interfaces for use in error conditions when the user may have only partial control of the system, or when they have no control due to SCRAM mode. Initially a presentation model of the alternate interfaces provides information about what operations are (and more crucially, are not) available for the user. Subsequently we can use the refinement theory based on μCharts trace refinement [2] to examine alternatives and prove that they are satisfactory.

We can also use the PIM of the new interface modes (safe, error_1, SCRAM) to ensure correct reachability, and via the Z representation of the PIM use model-checking to show that the correct modes are enabled given the defined safety parameters.

CONCLUSION

We have described the different models we use for interactive systems. We have the ability to combine these models into a single model which describes all parts of the interactive system. We can also use each of the models independently as the differing levels of abstraction mean that they are individually suitable for different tasks in the process of verifying and validating safety-critical interactive systems.

REFERENCES

1. J. Bowen and S. Reeves. 2006a. Formal Models for Informal GUI Designs. In *1st International Workshop on Formal Methods for Interactive Systems, Macau SAR China, 31 October 2006*, Vol. 183. Electronic Notes in Theoretical Computer Science, Elsevier, 57–72.
2. J. Bowen and S. Reeves. 2006b. Formal Refinement of Informal GUI Design Artefacts. In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)*. IEEE, 221–230.
3. Judy Bowen and Steve Reeves. 2013. Modelling Safety Properties of Interactive Medical Systems. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*. ACM, New York, NY, USA, 91–100. DOI: http://dx.doi.org/10.1145/2480296.2480314
4. Judy Bowen and Steve Reeves. 2014. A Simplified Z Semantics for Presentation Interaction Models. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. 148–162. DOI: http://dx.doi.org/10.1007/978-3-319-06410-9_11
5. Martin C. Henson, Moshe Deutsch, and Steve Reeves. 2008. *Z Logic and Its Applications*. Springer: Monographs in Theoretical Computer Science. An EATCS Series, 489–596.
6. ISO/IEC 13568. 2002. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics* (first ed.). ISO/IEC.
7. G. Reeve. 2005. *A Refinement Theory for μCharts*. Ph.D. Dissertation. The University of Waikato.
8. G. Reeve and S. Reeves. 2000. μ-Charts and Z: Hows, Whys, and Wherefores.. In *IFM*. 255–276.

Generating Domain-Specific Property Languages with ProMoBox: application to interactive systems

Bart Meyers
Modeling, Simulation and Design Lab (MSDL), University of Antwerp
Antwerp, Belgium
firstname.lastname@uantwerp.be

Romuald Deshayes, Tom Mens
Département d'Informatique, Université de Mons
Mons, Belgiumy
firstname.lastname@umons.ac.be

Hans Vangheluwe
Modeling, Simulation and Design Lab (MSDL), University of Antwerp / McGill University
Antwerp, Belgium / Montréal, Canada
firstname.lastname@uantwerp.be

ABSTRACT

Domain-Specific Modeling allows domain experts with limited technical background to precisely model applications by using domain concepts. These domain-specific models can be simulated, optimized, transformed into other formalisms, and from these models executable code and documentation can be generated. Because of their syntactic simplicity they are suitable for analysis, which is nonetheless often neglected in current approaches. Especially in Human-Computer Interaction, verifying whether the model satisfies its requirements (specified as so-called properties) is essential. The *ProMoBox* approach presents a highly automated solution for the specification and verification of such properties. It provides a framework for model checking of temporal properties, where all visible artifacts (system designs, properties, simulation traces, etc.) are specified in the domain-specific way.

THE *ProMoBox* APPROACH

Domain-specific modeling (DSM) helps designing systems at a higher level of abstraction. By providing languages, "DSMLs" (defined by a metamodel), that are closer to the problem domain than to the solution domain, low-level technical details can be hidden. An essential activity in DSM is the specification and verification of properties to increase the quality of the designed systems [3]. Providing support for these tasks is therefore necessary to provide a holistic DSM experience to domain engineers. Unfortunately, this has been mostly neglected by DSM approaches. At best, support is limited to translating models to formal representations on which properties are specified and evaluated with logic-based formalisms [6], such as Linear Temporal Logic (LTL). This contradicts the DSM philosophy as domain experts desiring to specify and verify domain-specific properties are not familiar with such formalisms. We propose the *ProMoBox* framework to shift property specification and verification tasks up to the DSM level. The scope, assumptions and limitations of this approach are presented in [5].

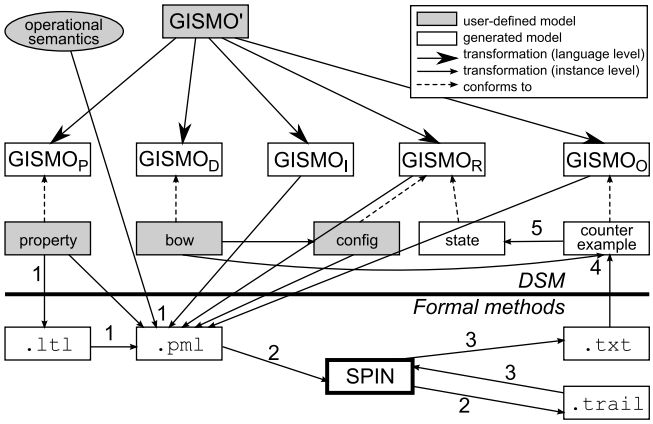

Figure 1. The *ProMoBox* approach applied to *GISMO*.

We applied *ProMoBox* to *GISMO* [1], a DSML for executable modeling of gestural interaction applications [2]. The *ProMoBox* approach for *GISMO* is illustrated in Fig. 1. The *ProMoBox* framework consists of (*i*) generic languages for modeling all artifacts that are needed for specifying and verifying properties, (*ii*) a fully automated method to specialize and integrate these generic languages in a given DSML, and (*iii*) a verification backbone based on model checking that is directly pluggable to DSM environments such as AToMPM [7]. Properties in *ProMoBox* are translated to LTL and a Promela model is generated that includes a translation of the system, its environment and its rule-based operational semantics. The Promela model is checked with the SPIN model checker [4] and if a counter-example is found it is translated back to the DSM level.

The *ProMoBox* framework [5] relies on a family of fully automatically generated modeling languages based on the DSML metamodel. These languages are required to modularly support specification and verification of model properties. The *design language* ($GISMO_D$ in Fig. 1) allows DSM engineers to design the static structure of the system. The *runtime lan-*

To cite this article use the following information:

Bart Meyers, Romuald Deshayes, Tom Mens, and Hans Vangheluwe. 2015. Generating Domain-Specific Property Languages with ProMoBox: application to interactive systems. In *Proceedings of the Workshop on Formal Methods in Human Computer Interaction* (FoMHCI'15), 47-48.
http://nbn-resolving.de/urn/resolver.pl?urn=urn:nbn:de:hbz:82-rwth-2015-030425

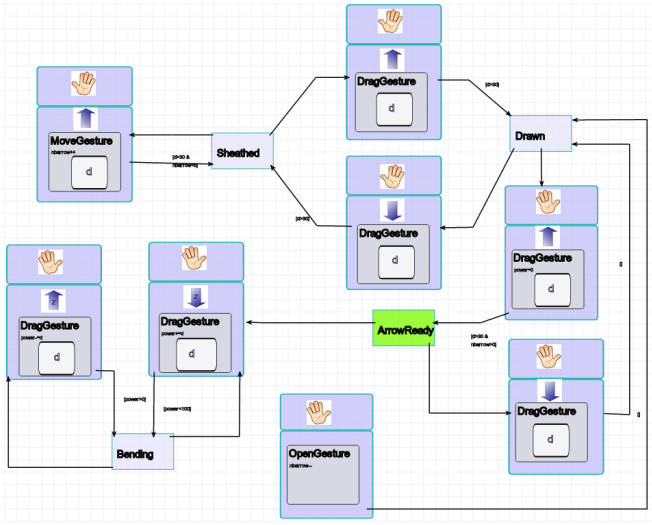

Figure 2. A bow model in state *ArrowReady* (highlighted) conform to $GISMO_R$.

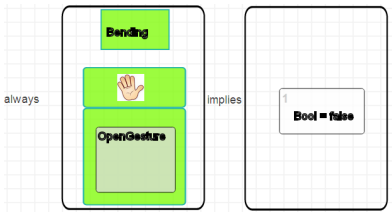

Figure 3. Property: when you fire the bow, there is no arrow left.

guage ($GISMO_R$) enables modelers to define a state of the system, *e.g.,* an initial state as input of a simulation, or a particular "snapshot" during runtime (as shown in Fig. 2). The *input language* ($GISMO_I$) lets the DSM engineer model the behavior of the system environment, *e.g.,* by modeling an input scenario as an ordered sequence of events containing one or more input elements. The *output language* ($GISMO_O$) can be used to represent execution traces (expressed as ordered sequences of states and transitions) of a simulation or to show verification results in the form of a counter-example. Output models can also be created manually as part of an oracle for a test case. The *property language* ($GISMO_P$) can be used to express properties based on modal temporal logic, including structural logic and quantification. A property is shown in Fig. 3.

Maintaining five DSMLs instead of one unacceptably increases the maintenance cost. Therefore, a fully automated method specializes and integrates these languages to any given DSML, thus minimizing the effort of the language engineer. This is realized by manually annotating the DSML metamodel entities (classes, associations and attributes) with the necessary UML-like stereotypes. This annotated metamodel (GISMO' in Fig. 1) contains all information needed to generate the five sublanguages, by merging a tailored version of the metamodel with a fixed template containing generic language constructs.

For fifteen properties, we verified whether the model shown in Fig. 2 satisfies them. The above properties are transformed to LTL, and are inserted in Promela code consisting of the system shown in Fig. 2 with initial state, the environment and rule-based model of the DSML's semantics as shown in step 1 of Fig. 1. In step 2, SPIN verifies whether the system satisfies the formula, returning "True" if it does. If there is a counter-example, steps 3 to 5 are followed: the counter-example trace is played back by SPIN, and a readable trace is printed (step 3), this trace is converted automatically to the counter-example output model (step 4), and this counter-example can be played out state by state by showing a runtime model for each state (step 5).

Because of these counter-examples, we were able to find and fix an error in our bow model of Fig. 2. In another instance, we were able to find and correct an error in one of the semantics model's rules. The performance in terms of time and memory consumption is good: evaluation never takes more than a second on an average laptop, and never requires more than 100 MB of memory.

The limitations of the framework are related to the mapping to Promela as explained in [5]. In its current state, *ProMoBox* does not allow dynamic structure models. Because of the nature of Promela, boundedness is ensured in the translation. Other constraints can be circumvented by abstracting the metamodel to make it suitable for model checking.

REFERENCES

1. R. Deshayes. 2013. A domain-specific modeling approach for gestural interaction. In *Visual Languages and Human-Centric Computing (VL/HCC)*. 181–182. DOI: http://dx.doi.org/10.1109/VLHCC.2013.6645275
2. Romuald Deshayes, Bart Meyers, Tom Mens, and Hans Vangheluwe. 2014. ProMoBox in Practice : A Case Study on the GISMO Domain-Specific Modelling Language. In *Proceedings of the 8th Workshop on Multi-Paradigm Modeling, MPM@MODELS 2014*. 21–30. http://ceur-ws.org/Vol-1237/paper3.pdf
3. Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 37–54. DOI: http://dx.doi.org/10.1109/FOSE.2007.14
4. Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (May 1997), 279–295. DOI: http://dx.doi.org/10.1109/32.588521
5. Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. 2014. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In *Software Language Engineering*. Lecture Notes in Computer Science, Vol. 8706. Springer International Publishing, 1–20. DOI: http://dx.doi.org/10.1007/978-3-319-11245-9_1
6. Matteo Risoldi. 2010. *A methodology for the development of complex domain-specific languages*. Ph.D. Dissertation. University of Geneva. http://archive-ouverte.unige.ch/unige:11842
7. Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. 2013. AToMPM: A Web-based Modeling Environment. In *MoDELS Demonstrations*. 21–25.