The MAL Interactors Animator: Supporting model validation through animation

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

Nuno Sousa
CCG – Centro de Computação Gráfica
Guimarães, Portugal
nuno.sousa@ccg.pt

ABSTRACT

The IVY workbench is a model checking based tool for the analysis of interactive system designs. Experience shows that there is a need to complement the analytic power of model checking with support for model validation and analysis of verification results. Animation of the model provides this support by allowing iterative exploration of its behaviour. This paper introduces a new model animation plugin for the IVY workbench. The plugin (AniMAL) complements the modelling and verification capabilities of IVY by providing users with the possibility to interact directly with the model.

CCS CONCEPTS

• **Human-centered computing** → **Human computer interaction (HCI)**; *Interaction design*; • **Software and its engineering** → **Software verification and validation**;

KEYWORDS

formal verification, model checking, model animation

ACM Reference Format:
José C. Campos and Nuno Sousa. 2018. The MAL Interactors Animator: Supporting model validation through animation. In *EICS '18: ACM SIGCHI Symposium on Engineering Interactive Computing Systems, June 19–22, 2018, Paris, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3220134.3220142

1 INTRODUCTION

The value of using modelling and automated reasoning tools to formally verify the design of systems is becoming clear. On the one hand, increasing complexity of the systems being developed makes it harder for designers and developers to maintain a complete and clear mental model of the design. On the other hand, systems are being developed to operate in more and more situations where failure has unacceptable consequences, creating the need for analysis techniques that provide as much assurance as possible. Interactive systems pose an additional challenge related to the need to factor in the human element during the analysis. However, the same style of approach has also been successfully applied.

Recent examples of using formal reasoning tools to analyse system designs include the use of model checking to support the safety analysis of the control logic for a neonatal haemodialysis machine [9], the user interfaces of components of a satellite launch system [1], clinical infusion devices [8] as well as the use of theorem proving in similar contexts, for example [10].

Using models to perform the analysis typically involves a number of steps:

(1) modelling the system — the model can be developed during design or reverse engineered from the implemented system;
(2) validating the model against the intended target system/design – modelling will typically be an iterative process of refining the intended or reverse engineered design, during this phase the model must be assessed against the intended design, both to prevent modelling errors and to perform a first validation of the design;
(3) specifying relevant requirements over the design (as properties of the model) — properties can be derived from system requirements, from applicable standards, or, in the case of interactive systems, from relevant usability guidelines;
(4) verifying that the model satisfies the properties — this will ideally be done with tools support; in the specific case of model checking, the analysis is fully automated and counter examples are provided when the property cannot be verified;
(5) analysing the verification results — when a property fails, a potential problem has been found and the results of the analysis must then be analysed; problems might relate to the system's design, to the model (e.g.

EICS '18, June 19–22, 2018, Paris, France

ACM ISBN 978-1-4503-5897-2/18/06…$15.00
https://doi.org/10.1145/3220134.3220142

Paper published as: J.C. Campos and N. Sousa (2018) The MAL Interactors Animator: Supporting model validation through animation. In Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pages 11:1-11:7. ACM.

due to the abstractions being used), or it might be found that other measures are in place that make the potential problem irrelevant.

Steps 2 and 5 both imply building an understanding (a mental model) of the system model. Model validation (step 2) can be achieved by exploring the system model. This can be done by challenging the model with properties that are known to be false of the design. The goal here is to establish that the model does not have undesirable behaviours, on the one hand, and to obtain valid behaviours that illustrate the system's function on the other hand. This however can be a cumbersome process. It is not easy to control which behaviours are generated by the analysis and it is not an approach that is easy to use with stakeholders. A more useful approach in this context is to animate the model. This makes it possible to control which behaviours to explore, which can ease the discovery of issues.

In the case of analysing verification results (step 5), the counter example produced by the model checker provides an example of a specific behaviour that falsifies the property. While this proves that a problem exists, it is not always the case that it supports an understanding of what the problem is, or why it happens. Again, the possibility of model animation is useful to support the exploration of the model around the highlighted problem.

IVY is a model based tool for the analysis of interactive systems. The tool has been used in the automotive domain [2], for space systems [1] and for medical devices [9]. IVY adopts a plugin based architecture to support experimentation with different functionalities and features. The base set of plugins includes: a model editor, a properties specification tool and a traces analyser. These plugins support steps 1, 3 and 4 above, but not steps 2 and 5.

The contribution of this paper is the description of the design and initial development of a animator plugin for the IVY workbench.

2 IVY WORKBENCH

The tool

Specifying a model in IVY is done textually using MAL interactors, a modal action logic based language [12]. Models adopt a production rules style, which has been found easy to understand by engineerings [6]. A textual editor plugin (the Models Editor – see Figure 1) is responsible for supporting model editing. Over the years, other alternatives have been experimented with, such as a graphical editor based on class diagrams or a tabular editor, but the textual editor remains the preferred alternative.

Verification is done through model checking, mapping the MAL interactors model to NuSMV [4]. Properties for verification are expressed in CTL (Computational Tree Logic) [5],

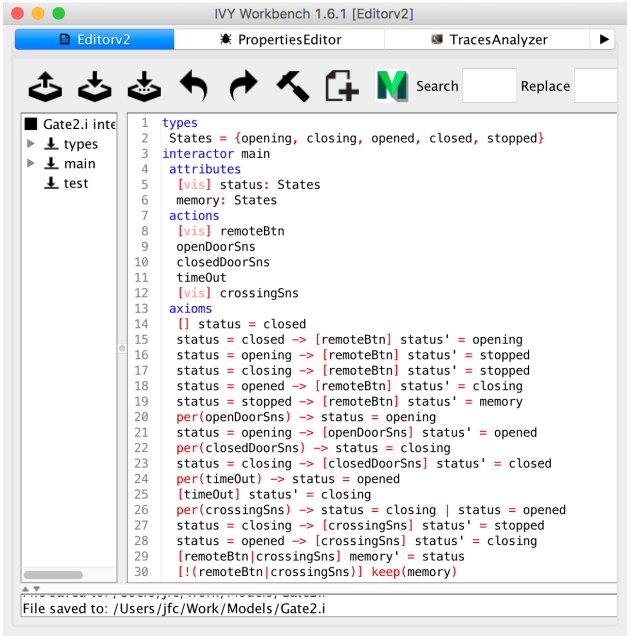

Figure 1: The IVY editor showing the example model

which supports expressing properties of the future behaviour of the system in a branching time model. A plugin (the Properties Editor) supports the definition of properties for verification. It uses the notion of property specification patterns [7] to facilitate the process of producing meaningful properties for verification. Once the model and the properties are translated to the NuSMV input language, verification is carried out by the model checker and results fed back to IVY.

Analysis of the verification results is supported by the Trace Analyser. When a property fails to check, a counter example (in the form of a trace – a behaviour of the model) is produced. The Trace analyser provides different representations of the trace (tabular, state based and activity based) as well as a mechanism to highlight states related to the conditions defined by the user of the tool.

An example model

The model of an automated gate will be used to illustrate the following discussion. The model, presented in Figure 1, is simple to bring clarity to the argument and to fit within the page limits of the paper. It is not representative of the complexity that can be addressed by the tool. More information about the IVY workbench, in general, can be found in [3].

The model consists of a single interactor (`main`) with two attributes: the `status` of the gate (one of `opening`, `closing`, `opened`, `closed` or `stopped`), and a memory of the system's previous status. The user can perceive the status of the gate, but not of the memory (hence the `[vis]` annotation

in the former). Five actions are possible. Pressing the button on the remote control (`remoteBtn`) and crossing the gate (`crossingSns`) are the only two actions available to the user. Two sensors detect when the door if fully open or closed (`openDoorSns` and `closedDorrSns`, respectively). Finally, the door will be closed automatically after a delay in the open state (`timeOut`).

The axioms define the initial state of the system (the gate is initially closed – line 14 in the model), the effect of the different actions under relevant conditions (if the door is closed, pressing the remote will cause it to start opening – line 15), and when actions are allowed (the triggering of the open door sensor is only possible if the door is opening – line 20).

The validation problem

As initially stated, once a model has been written, it must be validated. This step helps ensure that no obvious modelling errors have been committed and that the model captures the intended design. The goal is to determine if the model exhibits the expected behaviour, and thus gain confidence in its correctness with respect to the intended design (it must be noted that the model will be further challenged during the verification phase). This is typically achieved by attempting to prove that reaching desired goal states is not possible. If the model is correct, and reaching the goal state is possible, then a counter example, illustrating the correct behaviour of the system, is produced.

This approach, however, provides little flexibility in terms of exploring the behaviour of the system. Indeed, the system behaviour must be analysed indirectly, by causing the model checker to produce possible behaviours. Instead of directly, by interacting with the model. Additionally, the fact that the model checker produced a valid behaviour, does not necessarily mean that other, undesired, behaviours might not be possible, and vice-versa.

To address this issue, as well as to ease interpretation of verification results, a new plugin is being developed: the AniMAL (MAL models Animator) plugin. The role of the plugin is to support model validation by enabling direct interaction with the model. Additionally, it can be used to explore counter examples. While the trace analyser provides a static view of the produced trace, supporting querying, the animator support a dynamic analysis of the model, enabling the user to explore alternatives to the trace produced. The animator also uses NuSMV.

This new plugin will be described below, after relevant features of NuSMV have been briefly described.

3 NUSMV TRANSLATION AND NUSMV FEATURES

As stated above, MAL models are translated into NuSMV for verification. The MAL interactor language introduces a number of features that are not present in the target language, and which make MAL models more compact than the corresponding NuSMV model. The relevant ones in the present context are:

- MAL introduces the notion of action (cf. lines 7-12 in the model) – this is translated into a state attribute (`action`) in NuSMV. However, because actions can be parameterised, each action will involve as many values for the state attribute as the possible combinations of parameter values. However, all types must be finite for model checking with NuSMV to be feasible.
- MAL introduces the notion of action permission (cf. line 20) – this enable specification of when actions are allowed to happen; by default (in the absence of a permission axiom) actions are permitted. Permissions are translated into further transition rules at the NuSMV level, constraining the conditions under which the `action` attribute can take the value representing the action to those where the action is permitted.

Besides verification functionalities, NuSMV provides simulation capabilities. The relevant feature of NuSMV that enable the possibility of trace interactivity include [4]:

- the ability to choose one of the available initial states as the current state – a model might have more than one possible initial state and it possible to choose which state to use as the starting point for the trace.
- the ability to generate a sequence of n steps from the current state – the use of constraints supports the definition of what the next action will be. Setting the number of steps to one makes it possible to perform a step by step animation, where at each step the user will be able to choose the next action.
- the ability to navigate a trace – this supports backtracking a trace. This enables the exploration of alternative behaviours of the model.

At the moment, a NuSMV API is not available so interaction with the model checking tool is made possible by launching the NuSMV process and interacting with it through channels, simulating user input and then collecting output.

4 THE ANIMAL PLUGIN

This section describes the animator plugin (AniMAL). First requirements are identified.

Requirements

The accumulated experience of using IVY to analyse different systems has enabled us to identify a number of requirements for an animation plugin.

The basic goal of the tool is two-fold: Support for freely exploring the model; and support for understanding a specific behaviour of the model (a trace). The former is particularly

useful during model validation, while the latter during analysis of verification results.

Requirement 1 Users should be able to **observe the current state of the system** at each step in the animation – this is a basic requirement. A further refinement of this requirement is that the state should be presented in a manner consistent with other plugins of the IVY tool.

Requirement 2 Users should be able to **choose between valid actions** at each step in the process. The number of actions can grow considerably and not all actions are valid in every state, so only a valid action for the current state should be offered.

Requirement 3 Users should be able to **predict the effect of choosing an action**. There are two facets to this requirement. On the one hand, because the MAL models can be non-deterministic, it can happen that a single action might produce different results. To better support exploration of the model, the user should be able to choose which result to use. On the other hand, even if the model is deterministic, it is necessary to be able to predict what the effect of an action will be to support goal directed exploration of the model.

Requirement 4 User should be able to **observe the effect of the chosen action** in the model. While the first requirement is that users should be able to observe the state of the model at any given moment, here the goal is to support the analysis of how the model evolved over time.

Requirement 5 Users should be able to **explore all possible behaviours** of the model. Traces produced by model checking represent specific behaviours of the model. While they are helpful in illustrating *how* a property is not met, one limitation that was identified is that, by being fixed, they do not always help understand *why* the property is not met. The simulation component should bridge this gap by supporting the user in exploring all possible behaviours of the model.

Requirement 6 The plugin should be able to **replay traces**. This is a basic requirement to support the analysis of a particular trace, for example, a trace generated by a model checking attempt. Support for understanding a specific behaviour of the model (a trace) is required.

Features

To support the above requirements a number of features were designed and implemented.

Basic features. The user interface consists of two main areas (see Figure 2). The right hand side supports the analysis of the current state and past behaviour of the model (cf. Requirements 1 and 4). To comply with the requirement that states should be presented in a manner consistent with other plugins, trace representations provided by the Traces analyser plugin were adopted (traces are essentially sequences of states).

The plugin offers two trace representations: tabular, where states are presented as columns in a table (each line showing one attribute – see the table in Figure 2); and state based, a graphical representation where states are represented by boxes annotated with the state attributes (in this case each interactor is represented in a separate state sequence – see Figure 3). In both representations it is possible to see the current state of the model (cf. Requirement 1), as well as all states from the initial to the current one (cf. Requirement 4). Additionally, a log of the communication between the plugin and NuSMV is provided for development and maintenance purposes.

In Figure 2 the animation run has gone through three states (including the initial state), with the current state of the gate being `opened` and the memory having the value `closed`. The sequence of actions executed to reach this state from the initial (`closed`) state of the gate was: `remoteBtn` followed by `openDoorSns`.

The left hand side supports choosing the next action to execute. The actions shown are those that are valid for the current state of the model (cf. Requirement 2). Hence, in Figure 2 there are three possible actions from the current state: `crossingSns`, `remoteBtn` and `timeOut` (the nil action represents a stuttering step - i.e. the state of the interactor does not change; the action is relevant when the model contains more than one interactor; it is also the value of the `action` attribute in the initial state of the model).

To support Requirement 3, once an action is selected information about the values of the state attributes that will result from executing the action is provided. To support the user in the presence of non-deterministic models, if an action might have different outcomes, it is listed multiple times, one for each possible outcome of its execution. In the case of the figure, it can be seen that pressing the remote's button (action `remoteBtn`) will lead to a state where `status` and `memory` are `closing` and `opened`, respectively. No action appears repeated in the list so, in this state, the model is deterministic.

Support for Requirement 5 implies allowing the user to backtrack on the trace. This takes the model to a previous state where it is then possible to choose a different execution path by selecting a different action or the same action with a different outcome (in case of non-deterministic models).

The feature set above provides the basic functionalities to interact with a model. Below additional features are discussed that make this exploration more powerful.

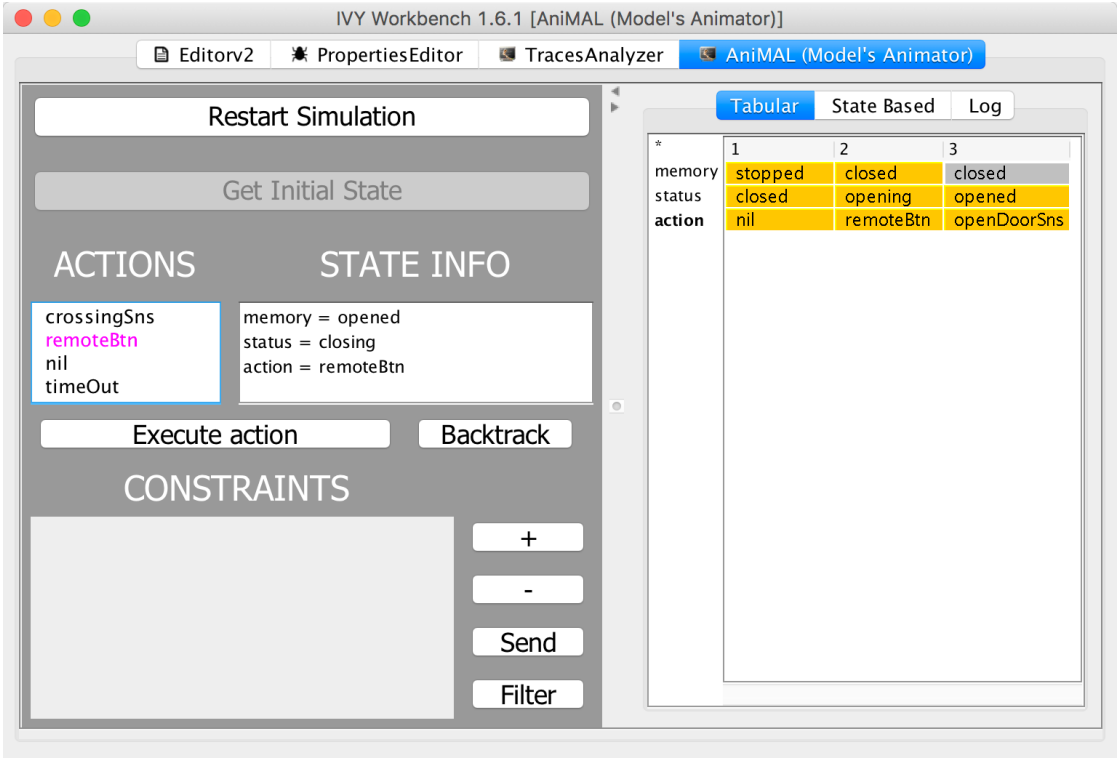

Figure 2: The plugin

Advanced features. When interacting with larger models (in particular in the presence of non-determinism) a problem might arise. In order to present only valid actions to the user, and to be able to present what the effect of those actions will be, the plugin makes a lookahead step in which it asks NuSMV to determine all possible next states. Using this information it then calculates the relevant information to present to the user. However, if the number of possible next states is too large, NuSMV will not calculate the states. In that case, constraints can be imposed in the simulation to reduce the state set that is generated. This can be done using the constraints dialogue (see bottom left of Figure 2).

Constraints consist of boolean expressions over the values of the state attributes or the actions in the model. They can be sent to NuSMV to constrain the next action execution. Additionally, they can be used to filter the current list of actions to better support choosing the relevant action to use next. In the latter case they affect only the display of the previously calculated list of action/results.

Finally, support for Requirement 6 implies allowing the user to load a trace and automatically run it. The user can then explore the model using the available features. The user will start by backtracking through the trace and then alternative actions will be made available to try.

5 DISCUSSION

The AniMAL plugin just presented enables IVY users to more easily gain an understanding of the model. This is useful when developing the model, to validate it against requirements, and particularly when analysing a model developed by a third party.

Besides helping gain this general understanding of the model, the animator has proven itself useful in a number of specific situations related with validating and debugging models. These have mainly to due with identifying aspects of the model which are either not sufficiently defined or constrained beyond what is needed.

A first case is identifying the **incomplete initialisation of the model**. If the values of the attributes in the initial state are not fully defined, then the model will have multiple possible initial states. Having more than one possible initial state for the model, while sometimes useful for verification purposes, can also create problems when interpreting verification results. In particular if the multiple initial states were not intended. The textual representation of the models does not make it easy to detect this situation; nor is it something that can be easily determined through verification. Using the animator it is straightforward to determine these situations

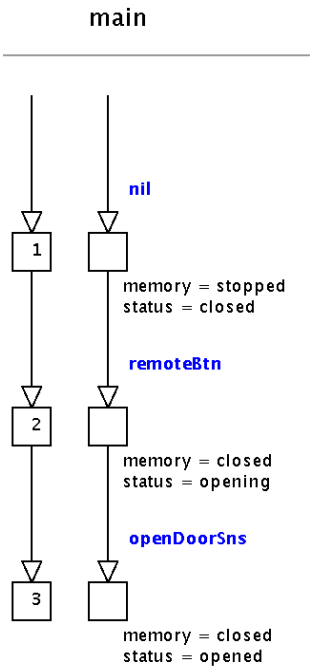

Figure 3: The state-based representation of the trace in Figure 2

as multiple initialisation steps (`nil` actions) are presented when starting the animation. In fact, for the model in Figure 2, five possible initial states exist, as the axiom in line 14 initialises the `status` attribute, but not the `memory` attribute.

A related case is identifying **incomplete action definitions**. If the axioms defining the effect of an action leave some of the state attributes undefined, then the model will have alternative behaviours, covering all the possible values allowed for those attributes. Using verification, these behaviours have to be spotted one by one. Using the animator these situations are more easily detected as the non-deterministic nature of actions becomes immediately evident by the fact that alternative outcomes for the action are offered. For example, if the axiom in line 30 was not present, the `timeout` action in the state depicted in Figure 2 would have five possible outcomes. One for each possible value of `memory` (while `status` would always be `closing` due to the axiom in line 25).

Conversely, identifying **stricter than intended action definitions**. The plugin supports identifying when the axioms defining an action do not allow a wanted behaviour. In that case, the desired result for the action will not be offered. The user can then investigate the axioms relevant to the action to identify the cause. The advantage is that the animator can be guided to relevant states, thus helping identify the conditions to use when analysing the axioms. Using the model checker it is not always the case that the trace highlights the problem state. One is left with a trace where the relevant action does not happen, or no trace at all, but no clear indication as of why.

Similarly, identifying **too strict permission axioms**. If the permission condition is stricter than needed, then the action will not be available in all situations it should. Determining the cause for this type of problem is particularly challenging using the model checker as the only information it provides is that the action cannot occur. No information about the reasons why is provided. These can be more easily identified using the animator as the action will not be offered when expected. As above, once a problematic state is identified, the user can then inspect the axioms to determine why.

Finally, identifying **too loose permissions** is also possible. If the permission condition is not strict enough, then the action will be possible when it should not be. Problematic states can be identified using the animator as the action will be offered when not expected.

While the animator by no means provides an exhaustive analysis of the model. Its goal is to support an initial validation, allowing more obvious problems to be quickly identified, and support the identification of the root causes of problems identified through verification. This makes it a valuable addition to the IVY workbench plugin set.

6 CONCLUSIONS

Formal verification is a powerful tool when designing complex systems. This is also true of interactive systems design. Model checking, in particular, supports a mostly automated analysis approach. Verification, however, does not always provide the flexibility needed to perform an initial validation of the model. It is also the case that analysis of the verification results could benefit from a iterative exploration of the model.

In this paper we have presented a new model animation plugin for the IVY workbench. The plugin (AniMAL) complements the modelling and verification capabilities of IVY by providing the possibility for its users to interact directly with the model. AniMAL is currently under development. All features are implemented except for the trace loading and the backtracking functionalities which are at the prototyping stage.

Future development includes using the plugin to run a prototype of the user interface. This will raise the abstraction level of the analysis and better support validating the model with stakeholders in the style of [11]. An initial version of this prototyping plugin is currently being developed.

ACKNOWLEDGMENTS

The authors wish to thank Michael D. Harrison for comments on an earlier version of this paper. José C. Campos acknowledges support from project NanoSTIMA (reference NORTE-01-0145-FEDER-000016) financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

REFERENCES

[1] J.C. Campos, M. Sousa, M. Alves, and M.D. Harrison. 2016. Formal Verification of a Space System's User Interface with the IVY workbench. *IEEE Transactions on Human-Machine Systems* 46, 2 (2016), 303–316. https://doi.org/10.1109/THMS.2015.2421511

[2] J. C. Campos and M. D. Harrison. 2008. Systematic analysis of control panel interfaces using formal tools. In *Interactive Systems: Design, Specification and Verification (Lecture Notes in Computer Science)*, Vol. 5136. Springer-Verlag, 72–85. https://doi.org/10.1007/978-3-540-70569-7_6

[3] J. C. Campos and M. D. Harrison. 2009. Interaction engineering using the IVY tool. In *ACM Symposium on Engineering Interactive Computing Systems (EICS 2009)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1570433.1570442

[4] R. Cavada, A. Cimatti, C.A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. 2010. *NuSMV 2.5 User Manual.* FBK-irst.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (April 1986), 244–263. https://doi.org/10.1145/5397.5399

[6] Martin B. Curry and Andrew F. Monk. 1995. Dialogue modelling of graphical user interfaces with a production system. *Behaviour & Information Technology* 14, 1 (1995), 41–55. https://doi.org/10.1080/01449299508914624

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. 1999. Patterns in Property Specifications for Finite-state Verification. In *Proc. 21st International Conference on Software Engineering (ICSE '99)*. ACM, 411–420. https://doi.org/10.1145/302405.302672

[8] M.D. Harrison, J.C. Campos, and P. Masci. 2015. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering* 11, 2 (June 2015), 95–111. https://doi.org/10.1007/s11334-013-0201-3

[9] M.D. Harrison, M. Drinnan, J.C. Campos, P. Masci, L. Freitas, C. di Maria, and M. Whitaker. 2017. Safety analysis of software components of a dialysis machine using model checking. In *Formal Aspects of Component Software (Lecture Notes in Computer Science)*, Vol. 10487. Springer, 137–154. https://doi.org/10.1007/978-3-319-68034-7_8

[10] M.D. Harrison, P. Masci, and J.C. Campos. accepted. Verification Templates for the Analysis of User Interface Software Design. *IEEE Transactions on Software Engineering* (accepted). https://doi.org/10.1109/TSE.2018.2804939

[11] P. Masci, P. Oladimeji, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby. 2015. PVSio-web 2.0: Joining PVS to HCI. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Vol. 9206. Springer, 470–478. https://doi.org/10.1007/978-3-319-21690-4_30

[12] M. Ryan, J. Fiadeiro, and T. Maibaum. 1991. Sharing Actions and Attributes in Modal Action Logic. In *Theoretical Aspects of Computer Software*. Lecture Notes in Computer Science, Vol. 526. Springer-Verlag, 569–593. https://doi.org/10.1007/3-540-54415-1_65