IVY 2 – A model-based analysis tool

Rui Couto
Dept. Informatics/University of Minho and
HASLab/INESC TEC
Braga, Portugal
rui.couto@di.uminho.pt

José Creissac Campos
Dept. Informatics/University of Minho and
HASLab/INESC TEC
Braga, Portugal
jose.campos@di.uminho.pt

ABSTRACT

The IVY workbench is a model-based tool that supports the formal verification of interactive computing systems. It adopts a plugin-based architecture to support a flexible development model. Over the years the chosen architectural solution revealed a number of limitations, resulting both from technological deprecation of some of the adopted solutions and a better understanding of the verification process to support. This paper presents the redesign and implementation of the original plugin infrastructure, originating a new version of the tool: IVY 2. It describes the limitations of the original solutions and the new architecture, which resorts to the Java module system in order to solve them.

CCS Concepts

•**Software and its engineering** → **Layered systems; Publish-subscribe / event-based architectures;** *Model checking;* •**Human-centered computing** → *User models; Interaction design theory, concepts and paradigms;*

Author Keywords

Model-based design and analysis; interactive systems; formal verification; model checking; tool support.

INTRODUCTION

Critical systems require an additional layer of quality assurance, when compared to traditional ones. A possible approach to support the verification of these systems is the use of model-based approaches. Systems are described by models, over which properties can be specified and verified. Counter-examples, produced when verification fails, provide insights into how the system should be redesigned.

IVY [4] is a framework that supports the verification of interactive systems, through a model-based verification approach. Systems are modeled in Model Action Logic (MAL) [15], which supports describing how available actions change the state of the system. The MAL model describes the structure of the systems (*attributes*), possible actions to be performed in the system (*actions*) and *rules* expressing system behaviour (*axioms*). The tool supports also the specification and verification of properties against the model, the analysis of generated counter-examples and interactive simulation of the model. Properties to be verified (*tests*) are expressed using CTL or LTL. A compiler translates the models into SMV specifications, which are verified with the NuSMV model checker [6]. When the verification fails, NuSMV produces traces that highlight sequences of states that invalidate the property. Visualization of these traces and interaction with the models are part of the IVY approach. It is possible to view verification results and interact with the models using different graphical state representations.

The tool adopts a plugin-based architecture, designed so that different solutions to support verification and validation might be explored. This has enabled a range of ideas to be explored, while maintaining a stable set of features in the tool. Nevertheless, over the years a number of issues was identified. First, development of the adopted, third-party, plugin framework stopped in 2007, meaning that its maintenance became harder to perform. Second, and related to the previous issue, when the architecture was first designed a centralized approach to information sharing was adopted. On the one hand, the adopted plugin framework did not provide an inter-plugin communication mechanism; on the other hand, plugins were seen as mostly independent tools. As a notion of analysis process matured, with plugins supporting different stages of the process, the need for a flexible inter-plugins communication mechanism became apparent. At the same time, it became clear that some services, such as model compilation or interaction with verification back-ends were needed across different plugins.

The main contribution of this paper is the description of a redesigned, plugins and services based, extensible framework, for the IVY tool, based in standard Java technology. The design of this framework results from the knowledge acquired during the several years developing, using and maintaining the first version of IVY. The paper can also help those interested in developing the tool.

THE VERIFICATION PROCESS

The verification of an interactive system can be decomposed into four main steps. In the first step a model of the system is created, typically describing both the structure and the behaviour of the user interface to be analysed. Depending on the stage of development, this model can be developed as an early representation of the interactive system's design, or might be reverse engineered from an existing implementation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EICS '19, June 18–21, 2019, Valencia, Spain

© 2019 ACM. ISBN 978-1-4503-6745-5/19/06…$15.00

DOI: https://doi.org/10.1145/3319499.3328228

Post-print of a paper published in the Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2019). Publisher's version available at: https://doi.org/10.1145/3319499.3328228

In the second step, the model is validated to check that it represents the intended system design. This might involve animating the model to observe its behaviour or proving basic properties that are expected to hold.

The verification of the model occurs in the third step. This step requires the identification of relevant verification requirements and their encoding in an appropriate logic. Requirements might come from, e.g., usability heuristics, regulatory documents, or a previous process of risk analysis. The analysis is then carried out, typically using a model checker or theorem prover.

In the final step, the results of the analysis are interpreted. In particular when the verification fails, it must be understood what the cause of the failure was and how it might be remedied.

THE IVY WORKBENCH

IVY is a model-based interactive computing systems analysis tool, based on the NuSMV model checker. The tool has successfully been applied in a number of different contexts, from aerospace systems [3] to medical devices [12].

IVY was developed in Java, which provides cross platform capabilities, resorting to the JPF[1] plug-in infrastructure library. Its plugin-based architecture enabled the development of a number of plugins to support the above verification process. A model editor plugin provides a text editor for the MAL language, supporting the task of creating the models. A patterns-based properties editor plugin supports the specification of properties for verification. This plugin uses the available compiler to generate a valid SMV model, and interacts with the NuSMV model checker to carry out the verification step. NuSMV is run as an external system process, as the model checker does not provide a communication API. A traces analyser plugin supports analyses of the verification results through the visualization of the counter-examples produced by NuSMV in a number of different representations. A models animator plugin supports interaction with the model and is useful both for the validation of the model and the analysis of verification results. It implements its own communication mechanism with NuSMV in order to perform the simulation (unlike for verification, for simulation purposes NuSMV must be run interactively).

Development of JPF, however, stopped in 2007 (supporting Java 5), meaning its use has became harder to maintain as the Java language and related tools evolve. Another related issue is that, while JPF provides the foundations for a plugin architecture, it does not enforce any particular structure. When IVY was first designed the goal was to provide as much flexibility as possible since, at the time, it was not completely known which features would be required in the architecture. For instance, except for a basic core set, we did not want to predetermine which plugins we would have and what kind of information they would share. As a consequence, no clear separation of concerns was enforced when the architecture was designed, and no inter-plugins communication mechanism was devised. The adopted solution was the placement of the in-memory representation of the model in a centralized information sharing mechanism, available through a core plugin. This created a separation between plugins and the data required to support them. With the adopted solution, all information transmission went through that core plugin.

The approach above resulted in a rigid solution. One the one hand, any new data type that might be required by a plugin must then exist in the centralized representation in order to be shared with others. On the other hand, the approach lacks a means of direct communication between plugins. This creates problems when dependencies between plugins exists. For example, it only makes sense to use the properties editor once a correct model has been edited, and to use the traces visualizer after the verification step. However, these plugins have no means to easily communicate with each other. They must rely on the information available centrally. One consequence of this is that changes to the shared data by one plugin might disrupt another plugin still working on the previous version of the data. In order to mitigate these dependencies, plugins started having their own representations of the required information. While solving the problem, this lead to repetition of code and information. For instance, there are representations of the model in the `CoreSystem`, which all plugins have access to, but also in the editor plugin and in the compiler, all with minor differences between them. As a consequence, changes in the MAL language require the three models to be updated accordingly.

The fact that no clear separation of concerns was defined, meant that the concept of standalone service was also not considered. Plugins were not expected to provide reusable features. Thus, functionalities such as the compiler exist in the core plugin, but interaction with the NuSMV model checker is implemented both in the properties editor plugin and the model animation plugin. While both plugins implement different interaction strategies, due to their different uses of NuSMV, a better solution would have been to develop the interaction feature as a service available to those plugins that needed it.

Overall, it is possible to say that the biggest limitation with IVY's original implementation was the lack of an inter-plugin communication mechanism. As for the developed core plugin, it presented a rigid solution as new features usually required changes in the core, which ultimately could affect existing plugins.

IVY VERSION 2

New plugins, exploring different ideas, are constantly being developed in IVY. Given the limitations identified in the preceding section, a new version of the plugin framework was developed, aimed at solving the identified issues and thus provide a solid architecture for the tool to evolve upon.

Objectives

Considering the aforementioned issues, a number of objectives were defined for the framework:

Objective 1 – Implementation of a clear model of inter-plugin communication.

[1] http://jpf.sourceforge.net, last visited March 24, 2019.

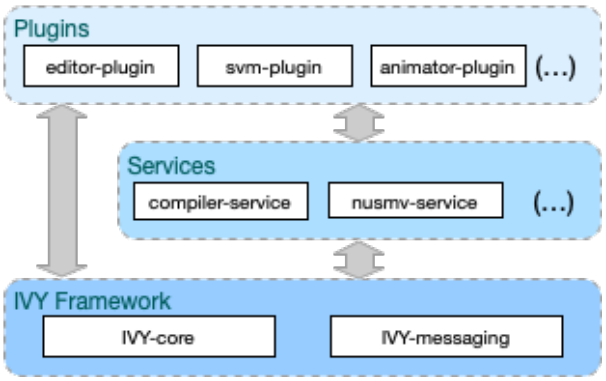

Figure 1. Base IVY architecture

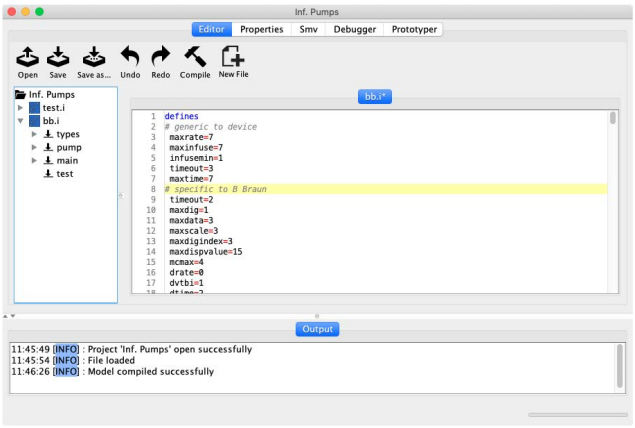

Figure 2. Example IVY plugins – the model editor

Objective 2 – A clear separation of concerns between *plugins* that implement features to be provided to users of the tool, and *services* that might be useful for different plugins.

Objective 3 – Support for plugin enabling/disabling without the need to recompile code.

Together, these objectives ensure improved development and maintenance processes of not only new and existing plugins, but also the framework itself. In particular it will become easier to develop new plugins, as detailed knowledge of the frameworks or other plugins or services implementations should not be needed. Only an understanding of the communication mechanisms and of the functionalities provided by existing plugins and services is necessary.

These objectives imply the following challenges:

Challenge 1 – How to provide an architecture separating core features from plugins, ideally without strong coupling.

Challenge 2 – How to guarantee support for the continued evolution of the framework.

The development of the new version IVY 2, depicted in Figure 1, had the objective of solving the identified issues, and achieve the defined objectives. This section describes both the decisions taken and the features developed.

Base architecture

Version 1 of IVY has shown that a plugin architecture is desirable. Based in previous experience, an approach similar to JPF would be adequate, but a more recent solution is required. We decided for the Java module system [16], and implemented the new version in Java 9.

IVY is composed of components of different nature. Some, such as the model editor (depicted in Figure 2) support user tasks, others as the compiler provide internal features. Thus, a three layer architecture was designed, as illustrated in Figure 1. At the bottom level (`IVY Framework`), the framework core is specified. It contains the core features, such as the basic user interface infrastructure, to which the plugins are added, and the messaging system. At the middle level (`Services`), we have the core functionalities of the application itself, containing services that provide functionalities such as compilation and verification. Finally, at the top layer (`Plugins`) there are the application end-user plugins, such as editors and visualizers.

While all components in the different levels correspond to Java modules, they are conceptually distinct. The core modules correspond to the framework itself, and while providing the framework's core features, they are not expected to suffer substantial changes, nor be affected by additional modules. Also, all other modules rely on these ones. Services are modules that provide core functionalities (such as compilation, simulation and verification), on which the plugins depend. Services are not expected to have an user interface. These components provide functionality reuse, something previously lacking in IVY. This is achieved by creating self-contained components which interact with the remaining ones through messages. Finally, plugins are the top level elements, which provide concrete functionalities to end users. The plugins are the elements that can be enabled or disable when launching IVY, thus supporting Objective 3.

This multi-layer architecture helps achieve Objective 2. By providing a separation between different features of the framework, when new developers want to integrate a feature, they can focus in the layers that concern their work. It is expected that developers will mainly focus in the top layer (`Plugins`), and understand some concepts of the middle layer (`Services`). From the bottom layer (`IVY framework`) developers need only to understand the communication mechanisms.

The IVY 2 plugin and service architecture is presented in Figure 3. Essentially. it provides the shared memory service (through the `SharedMemory` class), the `PluginService` interface that all plugins and services must implement (see Listing 1[2]), the `MessagingService` interface that again all plugins and services must implement in order to be notified of published messages (see Listing 2) and the `Messaging` class, which provides the API to publish messages.

Core communication features

The Java module system does not provide a framework by itself. While providing better organization of the different

[2]In the case of services, `getGUI()` must return `null`.

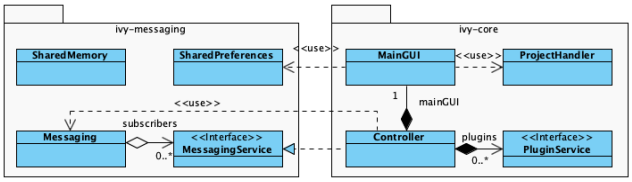

Figure 3. IVY 2 framework architecture

```
public interface PluginService {
    /** ... */
    public String getName();
    /** ... */
    public JPanel getGUI();
    /** ... */
    public int getPriority();
    /** ... */
    public void onCreate();
    /** ... */
    public void onFocus();
    /** ... */
    public boolean onFocusLost();
    /** ... */
    public void onDestroy();
    /** ... */
    public void onProjectSave();
    /** ... */
    public void onProjectOpen();
    /** ... */
    public List<JMenu> getMenuItems();
}
```

Listing 1. The API of a plugin

```
public interface MessagingService {
    /** ... */
    public void onMessageReceived(Object message);
    /** ... */
    public void onChannelMessageReceived(String channel,
                                         Object message);
    /** ... */
    public void onChannelMessageReceived(String channel,
                                         String message,
                                         Object data);
}
```

Listing 2. The messaging API

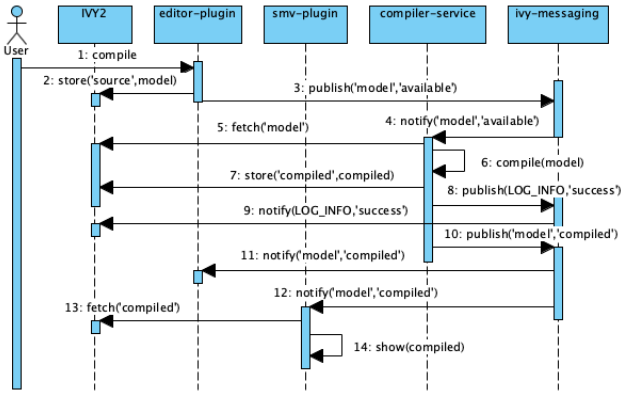

Figure 4. Example of workflow in IVY 2

parts of the framework (modules), other features are required to tackle remaining objectives and issues. In particular, in what relates to inter-plugin communication.

Messaging

A messaging system, implementing the publish-subscribe pattern [2], allows plugins to communicate without establishing strong dependencies between them. With this, we address Objective 3. The messaging system is part of the core framework. Plugins can use it to create their own channels, subscribe for notifications on existing channels, publish messages, and receive notifications when messages are published in the channels they have previously subscribed to. When a message is published in a channel, the framework will notify all plugins/services subscribing the channel by invoking the appropriate method (see Listing 2). There are three predefined channels, one to output log messages, another for providing progress information on activities that might take longer to carry out, and a third one to broadcast messages across all plugins. This mechanism has shown to successfully help remove previously existing dependencies and code repetition, allowing us to create independent services and plugins. For example, the compiler is no longer part of a plugin, but rather a service. The same is true for the interaction with NuSMV which was implemented in two different plugins and is now a service that both can use.

Shared memory

In order not to completely break away from the original model, a centralised key-value store is available for plugins to share information among them in a persistent manner. This shared memory further prevents coupling between plugins, as no direct references between them are required. A plugin can store a value in the store, which other plugins can consume. The mechanism provides the usual features to store information, verify if a certain key exists, and to retrieve information given a key.

The two communication mechanisms above can be combined. A plugin can publish some information (say, a model) in the store, and then issue a message announcing the information has been made available. Interested plugins will then be able to access it. This is illustrated in the next section.

Workflow

In order to help understand how the different modules in IVY integrate, this section illustrate a typical workflow of the tool. The section assumes an IVY installation where the editor and SMV[3] plugins, and the compiler service have been loaded and registered themselves in the `model` channel, where messages regarding the status of the models are published.

The workflow is that of compiling the model. The process (see Figure 4) starts in the model editor plugin, where the MAL model is described. Once the user selects the *Compile* action, the `editor-plugin` performs two actions: first, it stores the model in the shared memory, and then it sends a message to the `model` channel, informing that a new source code is available. Once the `compiler-service` receives the message, it retrieves the model from the shared memory, and proceeds to compile it. If the process is successfully performed (the case

[3] A helper plugin to display the generated SMV code.

depicted herein), the service sends a success message to the log channel[4], places the compiled result in the shared memory and broadcasts a message in the `model` channel, informing that a new compiled model is available. Finally, the `smv-plugin` retrieves the compiled model from the shared memory, after receiving the corresponding message, and displays it in its interface.

This simple example illustrates how the core components are integrated into a unified solution, which successfully allows the plugins and services to cooperate, without the need for strong coupling. Furthermore, it shows how the framework is able to support removing/adding plugins without affecting the existing solution. If the `smv-plugin` were to be removed, the only consequence would be the loss of the SMV code inspection functionality it provides. The compilation process would still be carried out between the editor and the compiler. Finally, the example helps illustrate that to develop new plugins and services, what is needed is only an understanding of the core services (messaging and shared memory) and of the basic plugin architecture.

DISCUSSION

The development of IVY 2 was based in prior experience: several years of development of IVY (version 1), feedback from different users, awareness of different problems due to a refactoring process, and ultimately the need to reduce maintenance costs. Based in previous experience, we designed a new architecture to both improve on the first version, and provide additional features. While the Java module system provides relevant features to better organize the source code, it does not constitute by itself a framework. In particular, we had to develop communication and information sharing mechanisms, that ultimately have shown to support our objectives and address Challenge 1.

Objective 3 was tackled by the modular design of the architecture, and specifically resorting the Java's standard modular architecture. Dividing the framework into three levels supports Objective 2, and reduces the knowledge required to start development in IVY 2. Objective 1 was met by the implementation of a publish-subscribe mechanism.

The adopted solution allowed us to progressively migrate the existing plugins into this new version, as existing plugins could be reused, if properly adapted to the new architecture. The migration process mainly consisted in moving the source code from IVY into IVY 2, and performing minor changes, in order to adjust to the new messaging system and comply with the new plugins API. The resulting architecture has also allowed us to develop new functionalities in existing plugins or new plugins and services in a much easier manner than was possible before. In particular, working in parallel is also easier. Four contributors were creating content in parallel, namely a visualizer, an editor and a wrapper for NuSMV, alongside with developments in the framework. Finally, this has shown also that integrating new developers was easier, as they need to focus mostly in the plugins they are developing. These results point towards tackling the second challenge.

[4]The framework will automatically display it in the output window, which can be seen at the bottom of Figure 2

RELATED WORK

Several tools exist that, like IVY, aim to support the formal verification of interactive computing systems. For example, tools such as PVSio-web [13], CIRCUS [9] or ADEPT [11]. Each tools has its specificities. A comparison of these tools falls outside of the scope of this paper. See, for example, [10], for a comparison of PVSio-web and CIRCUS. Here, we will focus on the technological choices that were made regarding IVY 2.

There are several existing approaches for developing modular architectures. We have previously used JPF, which while useful at the time, has seen its development stop, something that has raised the tool's maintenance cost. The Java module system has been integrated with the language itself, and became therefore a standard. OSGi is another popular [1] framework, which shares objectives with the Java module system. We have constrained the analysis of modular architectures to Java ones, since, first, we wanted cross-platform compatibility to exist, and second, we wanted to migrate the code from the previous version of IVY to the new version. We considered that using a standard technology is less prone to facing the problems previously faced with JPF, and thus chose the Java module system as the core for our implementation.

As previously stated, IVY relies in different tools to perform different tasks. Regarding the model editing capabilities, several approaches have been explored, such as the JEditorPane[5]. RSyntaxTextArea[6] is a library which has shown to be adequate due to its maturity and ease of costumizability, providing features such as search, history, and syntax highlight. There are several tools supporting model checking, a feature used by IVY to validate the models (see [14] for a comparison of ten popular ones). IVY currently resorts to NuSMV [6], but the new architecture will make it easier to integrate additional verification engines. At the moment the integration of support for nuXmv [5] is being considered, due to its improvements over NuSMV.

There are several possibilities to develop user interfaces in Java, the most popular being Swing [8] and JavaFX [7]. Each technology has its advantages and drawbacks. Swing is an easy to use standard library integrated into the Java framework. However, being its first version released in 1998, it lacks some features of modern frameworks, such as an integrated Model-View-Controller pattern. JavaFX was the proposal to solve Swing's problems and support modern interfaces. However, it was removed from the Java framework starting in version 11, and made into an open source framework. Looking forward for a more stable solution, Java Swing seems to be the best compromise. As a side note, we have explored the possibility of making IVY a web based application. However, we have decided that it was not the most adequate approach (given

[5]https://docs.oracle.com/javase/7/docs/api/javax/swing/JEditorPane.html, last visited March 25, 2019.

[6]http://bobbylight.github.io/RSyntaxTextArea/, last visited March 24, 2019.

the computational requirements imposed by the verification engines). This solution was thus disregarded for IVY 2.

CONCLUSIONS

The IVY workbench is a model-based tool that supports the verification of interactive computing systems. It supports an iterative process of modelling and analysis and provides the features to create and validate models, perform analysis through model checking and analyse verification results. The tool has been under active development for a number of years, with new features being experimented with, and existing ones being improved upon, as a better understanding of the process of interactive computing systems analysis through formal verification was obtained.

The tool adopted from the start a plugin-based architecture to better support this incremental and exploratory process of development. With time, however, the core plugin framework used was becoming hard to maintain and limited in its functionalities. As new developers took part in the process, it became harder for them to understand existing facilities, and correctly integrate new features. This drove the need to create a new version of the plugin infrastructure and of the tool as a whole: IVY 2.

This paper has reported on our experience in developing this new version, and documented the new framework, based on the Java module system. While the module system supported the organization of the code into pluggable units, the framework required additional features to standardise communication and the sharing of information between modules (plugins and services).

The current version of IVY 2 already mimics the functionalities available in the original tool. At the moment work is ongoing on both refining the framework and the integration of existing plugins, and on a set of additional plugins that include support for new functionalities such as prototyping or a tabular models editor to ease the learning curve of the tool.

ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project: UID/EEA/50014/2019.

REFERENCES

1. OSGi Alliance. 2003. *Osgi service platform, release 3*. IOS press.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley.
3. 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.
4. J. C. Campos and M. D. Harrison. 2008. Systematic analysis of control panel interfaces using formal tools. In *International Workshop on Design, Specification, and Verification of Interactive Systems*. Springer, 72–85.
5. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. 2014. The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification*. Springer, 334–342.
6. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. 2002. NuSMV 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*. Springer, 359–364.
7. J. Clarke, J. Connors, and E. J. Bruno. 2009. *JavaFX: Developing Rich Internet Applications*. Pearson Education.
8. R. Eckstein, M. Loy, and D. Wood. 1998. *Java Swing*. O'Reilly & Associates, Inc.
9. C. Fayollas, C. Martinie, P. Palanque, Y. Deleris, J.-C. Fabre, and D. Navarre. 2014. An approach for assessing the impact of dependability on usability: application to interactive cockpits. In *Tenth European Dependable Computing Conference*. IEEE, 198–209.
10. C. Fayollas, C. Martinie, P. Palanque, P. Masci, M.D. Harrison, J.C. Campos, and S. R. Silva. 2017. Evaluation of formal IDEs for human-machine interface design and analysis: the case of CIRCUS and PVSio-web. In *Third Workshop on Formal Integrated Development Environment (Electronic Proceedings in Theoretical Computer Science)*, Vol. 240. 1–19.
11. M. Feary. 2010. A Toolset for Supporting Iterative Human Automation: Interaction in Design. In *Selected Papers Presented at MODSIM World 2009 Conference and Expo*. NASA, 169–174.
12. M.D. Harrison, L. Freitas, M. Drinnan, J.C. Campos, P. Masci, C. di Maria, and M. Whitaker. 2019. Formal Techniques in the Safety Analysis of Software Components of a new Dialysis Machine. *Science of Computer Programming* 175 (April 2019), 17–34.
13. 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: 27th International Conference, CAV 2015, Part I*. Springer, 470–478.
14. F. Mazzanti and A. Ferrari. 2018. Ten Diverse Formal Models for a CBTC Automatic Train Supervision System. In *Third Workshop on Models for Formal Analysis of Real Systems and Sixth International Workshop on Verification and Program Transformation (Electronic Proceedings in Theoretical Computer Science)*, Vol. 268. 104–149.
15. 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, 569–593.
16. R. Strniša, P. Sewell, and Ma. Parkinson. 2007. The Java module system: core design and semantic definition. *ACM SIGPLAN Notices* 42, 10 (2007), 499–514.