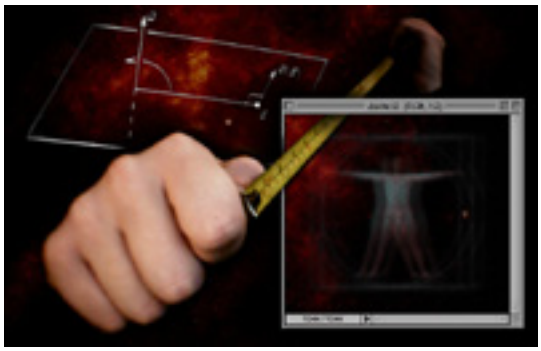
Crossroads
The ACM Student Magazine

User Interface Correctness

by ***Ian MacColl*** *and* ***David Carrington***

Introduction



A **user interface** is some boundary between a computer system, comprised of hardware and software, and a human user. User interface software is a significant component of contemporary computer systems, and graphical user interfaces are now almost universal.

Correctness of the user interface software is essential for critical applications, such as those affecting human life or safety. User interface correctness is also important for non-critical applications because the interface is the point at which the system is perceived by a user. Testing is one method that is used to assess software correctness, but testing user interface software is difficult.

This article is concerned with the use of formal methods to increase confidence in the correctness of user interface software. We start by considering notions of correctness and means of achieving it. We then introduce formal methods and outline how they can be used to increase confidence in the correctness of software. Finally, we survey the application of formal methods to the development of user interface software and indicate the potential for further work on specification-based testing of user interfaces.

What is correctness?

The IEEE Standard Glossary of Software Engineering Terminology defines **correctness**

in terms of freedom from faults, meeting of specified requirements, and meeting of user needs and expectations [20]. Software correctness is more commonly described as:

validation
: Are we building the correct software?

verification
: Are we building the software correctly?

Both validation and verification are important. Validation failure constitutes a breach of contract between the developer and the client for whom the software is being produced. Verification failure results in software containing potential faults or flaws. Clearly, neither is desirable.

Correctness is particularly important for software which is critical in some way, for instance when human lives are at risk. We would expect that correctness of all aspects of such software, including the user interface, would be strenuously pursued. This is not necessarily the case. The widely cited Therac-25 accidents, which resulted in deaths and injury, are partly attributed to enhancing usability of the user interface at the expense of the safety of the overall system [21].

Correctness is important for software in general, however, and is particularly important for user interface software. The user interface represents the aspect of the software which is directly perceived by a user. If the user interface is incorrect, the software will be perceived as incorrect, regardless of the correctness of the underlying functionality.

Confidence in the correctness of a user interface is usually achieved by prototyping or by testing. Prototyping can be used for validation and to verify that the user interface meets usability requirements. Testing can also be used for validation (acceptance test) and to evaluate usability. Traditional software testing (unit, integration, and system test) is the major method for assessing the correctness of user interface software.

Although testing of user interface software, particularly graphical user interface software, has elements in common with other software testing, it also presents a number of challenges:

- The user interface is a large and complex component of a software system.
- Graphical and other presentations make it difficult to determine the expected

result of an operation, particularly in the context of system-wide environment settings such as colors, desktop graphics and default fonts.
- Input is derived from multiple, asynchronous devices, usually at least a mouse and a keyboard.
- Interaction styles are expected to be modeless with enabled actions permitted (and vice versa).
- Rapid semantic feedback on the results of operations is expected.

In summary, user interface software is complex, highly interactive, modeless, concurrent, graphical, and has user-based real-time requirements. Emerging user interface technologies such as multi-modal interaction, multi-media, intelligent agents, and ubiquitous computing and communications will increase the testing problems.

Industrial strength tools have been developed to meet these challenges. These tools provide facilities such as capture/playback for test development, bitmap- and widget-based test evaluation, and textual and point-and-click scripting. Although they provide essential support for the testing of user interface software, little support is available for determining what tests to perform and, having decided on tests, for deciding what output to expect from a particular test.

The definitions at the start of this section indicate that correctness is determined with respect to some expectation or specified requirement. Test selection and evaluation can also be based on such information. Formal methods provide a means for defining such information and are the subject of the next section.

What are formal methods?

Formal methods (http://www.comlab.ox.ac.uk/archive/formal-methods.html) are ``mathematically based techniques for describing system properties'' [**33**]. They typically include a precise notation for constructing mathematical models of a (software) system. The notation is used for specifying (rather than designing or implementing) the required system, and is concerned with ``what'' is done by a system rather than ``how'' it is done. Two defining characteristics of formal methods are *precision* and *abstraction*.

Many formal methods use a notation with a mathematical appearance, but this is not a requirement. For example, programming languages with a sound semantic basis can be considered to be specification notations; they are precise notations for constructing

(executable) models of a system (although they are not particularly abstract). Well-defined graphical notations such as **Petri Nets (http://www.daimi.aau.dk/PetriNets)** and Harel's Statecharts are also specification notations.

Formal methods can be classified as model-, property-, or behavior-based. Model- and property-based methods are concerned with modeling underlying data structures and their operations whereas behavior-based methods focus on externally observable behavior.

Model-based methods define system behavior directly by constructing a model using mathematical structures such as sets. Model-based notations for sequential systems include **Z** (pronounced ``Zed''; http://www.comlab.ox.ac.uk/archive/z.html) and **VDM (http://www.ifad.dk/vdm/vdm.html)**. Such notations are suited to specifying software involving complex data structures and simple operations since the data types are modeled explicitly.

Property-based notations such as **OBJ (http://www.comlab.ox.ac.uk/archive/obj.html)** and **Larch (http://larch-www.lcs.mit.edu:8001/larch/)** model data types implicitly by defining system behavior through a set of properties. Property-based methods can be further classified as *axiomatic* and *algebraic*: axiomatic methods are based on first-order predicate logic and algebraic methods use equational axioms.

Behavior-based methods define systems in terms of possible sequences of states rather than data types and are commonly used to specify concurrent and distributed systems. Behavior-based notations include **Petri Nets**, **Calculus of Communicating Systems (CCS) (http://ei.cs.vt.edu/~cs5204/ccs.html)**, **Communicating Sequential Processes (CSP) (http://www.comlab.ox.ac.uk/archive/csp.html)**, and Statecharts.

Formal methods can be used to increase confidence in the correctness of software by proof, refinement and testing. **Proof**, sometimes called formal verification, involves a rigorous demonstration (usually involving deductive logic) that an implementation matches its specification. **Refinement** is the development of implementations that are correct by construction; a specification is rigorously transformed to derive an efficient implementation. **Testing** involves executing an implementation with some input and comparing the resulting output to output expected for that input. Specification-based

testing uses a (formal) specification for determining appropriate inputs and expected outputs.

Formal methods have been advocated for applications where the cost of software failure is greater than the presumed additional cost of the formal development process, for example, for mission-, safety-, and life-critical software. Industrial experience indicates, however, that additional costs (for example, the training of personnel to use the formal methods) are more than offset by other savings.

- Z was used to specify 14% of IBM's CICS/ESA 3.1 release and the total development cost was reduced by 9% [**25**].
- IBM's **Cleanroom (http://www.clearlake.ibm.com/MFG/solutions/cleanrm.html)** software engineering process combines formal methods (specification and proof) with statistical testing to improve quality and productivity; error rates an order of magnitude better than industry averages have been reported and lines of code per person-month metrics have been 36% better than industry averages [**15**].
- Hall reports on development of a large air traffic control information system using a combination of formal methods. The system was large (197,000 lines of C code) with significant real-time and safety requirements; overall productivity rates were comparable to or better than industry averages and error rates were significantly better than industry averages [**12**].

Hall also published a seminal article in 1990 identifying and dispelling popular misconceptions about formal methods [**11**]. His first myth, ``Formal methods can guarantee that software is perfect'' amounts to a claim that formal methods can eliminate software testing. He rebuts this myth by noting that ``Some things can never be proved and we can make mistakes in the proofs of those things we can prove.''

Hall's view is reinforced by Bowen and Hinchey in their seventh `commandment' for successful formal methods projects: ``Thou shalt test, test and test again'' [**1**]. A formal specification is an abstract representation that can be rigorously transformed (refined) to an executable implementation. The transformation process cannot be fully automated and testing can indicate the possibility of (human) error. Testing can also be used in fully formal development to detect faults in the compiler or operating system used to execute the software.

Software testing is conventionally classified into behavioral and structural testing: *behavioral testing* emphasizes testing against the functional requirements implemented by the software while *structural testing* emphasizes testing based on the internal structure of the software design and implementation.

Structural testing is primarily used to determine whether all parts of the software have been tested; behavioral testing is used to assess software correctness. The precise definition of behavior provided by a formal specification is particularly useful for behavioral testing as a basis for deriving test inputs and as a means for determining the output expected from a particular test input.

The application of formal methods to testing, also called *specification-based testing*, is an active research topic. Gaudel, for example, summarizes program testing based on algebraic specifications, describing methods for selecting a finite subset of an exhaustive test set [**9**]. For model-based notations, Hörcher and Peleska show how Z specifications can be used to derive test input data and to automatically evaluate test results [**17**].

The Test Template Framework and the ClassBench methodology are testing techniques under investigation at The University of Queensland, Australia. The Test Template Framework is a formal, abstract model of testing, used to derive a hierarchy of test information, including test inputs and outputs, from a formal specification [**2**, **29**, **30**]. The ClassBench Methodology and the Test Template Framework provide an approach to automated testing of object-oriented software [**16**]. The Specification-Based Testing project, funded by the Australian Research Council and based at the Software Verification Research Centre, involves researchers from Rutgers University, USA, and University of Victoria, Canada, and aims to combine Test Templates and ClassBench to create methods and tools for testing object-oriented software.

This section has introduced formal methods and outlined how they can be used to improve confidence in the correctness of software. The next section is a survey of the application of formal methods to the development of user interface software and indicates the potential for further work on specification-based testing of user interfaces.

How are formal methods applied to HCI?

Formal methods have been applied to the development of user interface software with two aims: to abstract from the details of users, software, and their interaction a basis for reasoning and analysis, and to ensure correct implementation of the required software. The research literature appears dominated by work concerned with formal analysis of HCI. This paper is concerned with the use of formal methods to achieve correct user interfaces.

Took identifies two paradigms (or architectures) for user interface software: linguistic and agent-based [**32**]. The linguistic paradigm is based on the work of Foley and van Dam who distinguish lexical, syntactic, and semantic aspects of interactive command languages. Formal notations for the linguistic paradigm are typically syntactically based and behavior-oriented, like Backus-Naur Form (BNF) (**http://phrantic.com/j_alan/comp2.html#EBNF**) [**26**] or state-transition diagrams [**24**], and are not well suited to handling concurrency or to attaching semantic information.

Agent-based architectures are an attempt to resolve the limitations of linguistic architectures as a basis for developing graphical user interfaces. The agent-based paradigm encapsulates data, functionality, and input and output within an `agent' such as a button, a screen, or an entire user interface. The encapsulation is consistent with abstract data type and object-oriented software development.

Application of formal methods to agent-based architectures is a current research topic. Like object-oriented software development, formal approaches to development of agent-based user interfaces must address both static and dynamic aspects of the system. Model- and property-based notations are well-suited to formalizing static aspects, whereas behavior-based notations are preferable for dynamic aspects. Many formal, agent-based approaches use multiple notations of different styles, or extend an existing notation to encompass capabilities of a different style.

Behavior-based notations, such as Petri Nets, are useful for modeling external behavior but must be augmented to include static aspects of user interfaces. Palanque and Bastide, for example, use the Interactive Cooperative Objects (ICO) formalism to specify, verify, and implement user interfaces [**23**]. ICO provides an object-oriented framework for modeling static aspects of a system, and uses Petri Nets for modeling dynamic aspects.

Myers introduces a model for ``highly interactive, direct manipulation, graphical user

interfaces'' that encapsulates interaction into *interactor* objects [**22**]. The Esprit AMODEUS project has developed two formal models of interactors as encapsulations of a state, events that manipulate the state, and a mechanism to present the state to users [**4**, **3**, **13**]. The first interactor model, developed at CNUCE in Pisa, Italy, uses the process algebra **LOTOS (http://wwwtios.cs.utwente.nl/lotos)**, a property-based notation with behavioral capabilities. The CNUCE model is derived from work on graphics systems and input devices.

The second AMODEUS interactor model was developed at University of York, UK. The York model uses model-based notations such as Z and VDM, augmented by CSP for behavioral aspects. The York model is a development of the work of Sufrin and He [**31**] extending Z to cover both static and dynamic aspects. The AMODEUS project has used interactor models for describing graphics systems and for analyzing interactive systems. Current work includes integration of formal descriptions of users and systems in an interactive framework, and investigation of scaling up formal development of user interface implementations.

Much of the work applying formal methods to HCI emphasizes reasoning and analysis rather than implementation. A current project at The University of Queensland [**18**, **19**] is concerned with formal notations for (object-oriented) specifications of user interface software, and the development of designs by using relatively informal transformations based on software patterns [**8**]. This project is using **Object-Z (http://www.cs.uq.edu.au/svrc/Object-Z)** [**27**, **6**], an object-oriented extension of Z, for interactor specifications. Object-Z enhances the model-based capabilities of Z with encapsulation constructs, and includes operators for expressing dynamic aspects such as concurrency and communication between objects.

In addition to formally-based development, testing was identified above as a means of improving confidence in the correctness of software. To date, the work of Yip and Robson appears to be the only use of formal methods to directly support testing user interface software [**34**, **35**]. They use three notations to specify user interface software: state-transition diagrams to express interaction sequences, a model-based notation similar to Z to define the state transitions, and an algebraic notation for reasoning. Tests are derived from the state-transition diagrams, and the model-based specifications are used to determine expected outputs.

A variation of formally-based testing is the use of constraints in the implementation.

Fisher and Frincke augment an algebraic specification notation with display axioms that connect abstract application operations with concrete interface operations [**7**]. The axioms are used to verify interface correctness with respect to underlying functionality and they are executable to maintain the correspondence of interface and functionality at runtime. Boolean pre- and post-conditions are used for a similar purpose by Gieskins and Foley [**10**]. In effect, the conditions are a specification and they are used to control visibility and enabling, for prototyping and for generating documentation.

Conclusions

In this article we have discussed user interface correctness and outlined ways it can be achieved through formal methods. There is still considerable work remaining to be done in applying formal methods to develop correct user interface software.

Scaling up formal development techniques, such as refinement, for the production of industrial-strength user interfaces is an open issue. One particularly interesting aspect involves consideration of decomposition and refinement. Duke and Harrison [**5**] show that an abstract specification *Abs* can be refined into separate specifications of the user interface *Usr* and the application functionality *Fun*, and that *Fun* is a refinement of *Usr* (i.e., the user interface is an abstraction of the functionality). This raises questions for the overall software development process: refinement of *Abs* to *Usr* to *Fun* emphasizes a user-oriented design approach whereas refinement of *Abs* to *Fun* followed by derivation of *Usr* from *Fun* by abstraction emphasizes system-oriented design concerns. The desirability of iterative development and the use of multiple notations with differing notions of refinement adds further complexity to this issue.

The lack of work on specification-based testing of user interfaces is not particularly surprising given the relative youth of these two areas. Specification-based testing can provide an intermediate use of formal methods beyond simple specification, without the costs (and benefits) of a fully formal development process. Existing specification-based techniques target particular specification styles so that application of these techniques must consider integration of multiple notations to cover both static and dynamic aspects. The Test Template Framework, for example, is well suited to specification-based testing of static aspects of model-based user interface specifications but is less suitable for testing dynamic aspects. The Specification-Based Testing project will address testing of dynamic aspects of object-oriented formal specifications but is likely to focus on the Object-Z notation. In our own research, we

are investigating options for extending the Test Template Framework to accommodate multiple notations or perspectives to support specification-based testing of user interfaces.

Acknowledgements

We thank Anna Andrusiewicz and Anthony MacDonald for reading a previous draft of the paper. Ian MacColl is supported by an Australian Postgraduate Award and by a Telstra Research Laboratory Postgraduate Fellowship.

References

1

Bowen, Jonathan P. and Hinchey, Michael G. Ten commandments of formal methods. *IEEE Software*, 28(4):56-63, April 1995.

2

Carrington, D. and Stocks, P. A tale of two paradigms: Formal methods and software testing. In J. E. Nicholls and J. A. Hall, editors, *Z User Workshop*, pages 51-68. Springer-Verlag, Cambridge, June 1994.
Also published as Technical Report 94-4, Software Verification Research Centre, Department of Computer Science, The University of Queensland.

3

Duke, D., Faconti, G., Harrison, M., and Paterno, F. Unifying views of interactors. In *Proceedings of the Workshop on Advanced Visual Interfaces*, pages 143-152. ACM Press, June 1994.

4

Duke, D. and Harrison, M. Abstract interaction objects. *Computer Graphics Forum*, 12:25-36, 1993.

5

Duke, D. J., and Harrison, M. D. Mapping user requirements to implementations. *IEE/BCS Software Engineering Journal*, 10(1):13-20, January 1995.

6

Duke, R., Rose, G., and Smith, G. Object-Z: A specification language advocated for the description of standards. *Computer Standards & Interfaces*, 17:511-533, 1995.

7

Fisher, G. L., and Frincke, D. A. Formal specification and verification of graphical user interfaces. In Shriver [**28**], pages 114-123.

8

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. ***Design Patterns: Elements of Reusable Object-Oriented Software***. Addison-Wesley, 1994.

9

Gaudel, M. Testing can be formal, too. In *TAPSOFT '95 : Theory and practice of software development : Sixth International Joint Conference on Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82-96, May 1995. Aarhus, Denmark.

10

Gieskens, D. F., and Foley, J. D. Controlling user interface objects through pre- and postconditions. In Penny Bauersfeld, John Bennett, and Gene Lynch, editors, *Striking A Balance: Conference on Human Factors in Computing Systems (CHI '92)*, pages 189-194, Monterey, CA, May 1992.

11

Hall, A. Seven myths of formal methods. *IEEE Software*, 7(5):11-19, September 1990.

12

Hall, A. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66-76, March 1996.

13

Harrison, M. D., and Duke, D. J. A review of formalisms for describing interactive behaviour. In *Software Engineering and Human-Computer Interaction*, volume 896 of *Lecture Notes in Computer Science*, pages 49-75. Springer-Verlag, 1995.

14

Harrison, M., and Thimbleby, H., editors. ***Formal Methods in Human-Computer Interaction.*** Cambridge University Press, 1990.

15

Hausler, P. A., Linger, R. C., and Trammell, C. J. Adopting Cleanroom software engineering with a phased approach. *IBM Systems Journal*, 33(1):89-109, 1994.

16

Hoffman, D., and Strooper, P. The testgraph methodology: Automated testing of collection classes. *Journal of Object-Oriented Programming*, 8(7):35-41, November-December 1995.

17

Hörcher, H., and Peleska, J. Using formal specifications to support software testing. *Software Quality Journal*, 4(4):309-327, December 1995.

18

Hussey, A., and Carrington, D. Using Object-Z to compare the MVC and PAC architectures. In C. R. Roast and J. I. Siddiqi, editors, *Proceedings of BCS-FACS Workshop: Formal Aspects of the Human Computer Interface*, pages 45-60, Sheffield, UK, September 1996.
Also published in longer form as Technical Report 95-33, Software Verification Research Centre, Department of Computer Science, The University of

Queensland.

19

Hussey, A., and Carrington, D. Using Object-Z to specify a web browser interface. In John Grundy and Mark Apperley, editors, *Proceedings OzCHI*, pages 236-243, Auckland, New Zealand, November 1996.
Also published as Technical Report 96-6, Software Verification Research Centre, Department of Computer Science, The University of Queensland.

20

IEEE. Standard Glossary of Software Engineering Terminology. In *IEEE Software Engineering Standards Collection*. IEEE, 1994. Std 610.12-190.

21

Leveson, N. G., and Turner, C. S. An investigation of the Therac-25 accidents. *Computer*, 26(7):18-41, July 1993.

22

Myers, B. A. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289-320, 1990.

23

Palanque, P., and Bastide, R. Petri Net based design of user-driven interfaces using the Interactive Cooperative Objects formalism. In Fabio Paterno, editor, *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems*, pages 383-300, Bocca di Magra, Italy, June 1994.

24

Parnas, D. L. On the use of transition diagrams in the design of a user interface. In *Proceedings of the 1969 National ACM Conference*, pages 739-743. Association for Computing Machinery, 1969.

25

Phillips, M. CICS/ESA 3.1 experiences. In J. E. Nicholls and J. A. Hall, editors, *Z User Workshop*, pages 179-185, Oxford, December 1990. Springer-Verlag. Proceedings of the Fourth Annual Z User Meeting.

26

Reisner, P. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering*, SE-7:229-240, March 1981.

27

Rose, G. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, ***Object Orientation in Z***, Workshops in Computing, chapter 6, pages 59-77. Springer-Verlag, 1992.

28

Shriver, B. D., editor. *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, 1991.

29

Stocks, P., and Carrington, D. Test templates: A specification-based testing framework. In *Proceedings International Conference on Software Engineering (ICSE)*, volume 15th, pages 405-414. IEEE Computer Society Press, Baltimore, MD, May 1993.
Also published in a longer form as Technical Report 243, Department of Computer Science, The University of Queensland.

30

Stocks, P., and Carrington, D. Test Template Framework: A specification-based testing case study. In Thomas Ostrand and Elaine Weyucker, editors, *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 11-18. ACM Press, June 1993.
Also published in a longer form as Technical Report 255, Department of Computer Science, The University of Queensland.

31

Sufrin, B., and He, J. Specification, analysis and refinement of interactive processes. In Harrison and Thimbleby [**14**], chapter 6, pages 153-200.

32

Took, R. Putting design into practice: Formal specification and the user interface. In Harrison and Thimbleby [**14**], chapter 3, pages 63-96.

33

Wing, J. M. Formal methods. In John J. Marciniak, editor, ***Encyclopedia of Software Engineering***, pages 504-517. John Wiley & Sons, 1994.

34

Yip, S. W. L., and Robson, D. J. Graphical user interfaces validation: A problem analysis and a strategy to solution. In Shriver [**28**], pages 91-101.

35

Yip, S. W. L., and Robson, D. J. Window user interfaces and software maintenance. *Software Maintenance: Research and Practice*, 3:107-123, 1991.

Ian MacColl (http://www.cs.uq.edu.au/personal/ianm) (ianm@cs.uq.edu.au) is a doctoral student in the Software Verification Research Centre of the Computer Science Department of The University of Queensland, Australia. His research interests include formal methods, user interfaces, software testing and object-oriented and functional programming.

David Carrington (http://www.cs.uq.edu.au/personal/davec) (davec@cs.uq.edu.au) is a senior lecturer in the Department of Computer Science and an academic member of the Software Verification Research Centre at The University of Queensland,

Australia. As well as his interest in teaching software engineering, David's research interests focus on formal approaches to software and user interface development. This includes research into both methods and supporting tools.