# Test of the ICARE platform fusion mechanism *

S. Dupuy-Chessa[1] and L. du Bousquet[2] and J. Bouchet[1] and Y. Ledru[2]

[1] CLIPS-IMAG, BP 38, 38041 Saint Martin d'Hères cedex 9, France
[2] LSR-IMAG, BP 72, 38402 Saint Martin d'Hères cedex 2, France
E-mail: {sophie.dupuy,lydie.du-bousquet,jullien.bouchet,yves.ledru}@imag.fr

**Abstract.** Multimodal interactive systems offer a flexibility of interaction that increases their complexity. ICARE is a component-based approach to specify and develop multimodal interfaces using a fusion mechanism in a modality independent way. ICARE being reused to produce several multimodal applications, we want to ensure the correctness of its fusion mechanism. So we validate it using a test architecture based on Java technologies. This paper presents our validation approach, its results, its advantges and its limits.
**Keywords :** test, multimodality, fusion mechanism

## 1 Introduction

An increasing number of applications in the field of human computer interfaces support multiple interactions such as the synergistic use of speech, gesture and eye gaze tracking. Multimodal applications are now being built in different application domain including medecine [15], military [5] or telecommunication [12]. For example, lot of mobile phones offer two input modalities, a keyboard and a voice recognizer, to interact. Although many multimodal applications have been built, their development still remains a difficult task. The major difficulty concerns the technical problem of the fusion mechanism, such as the application of R. Bolt [1], in which a speech modality "put that there" is blending with a gesture modality, specifying positions defined by the deictic "that" and "there". Here, the difficulty is to blend correctly the several data from the modalities. Thus, the fusion algorithm must be rigorously and carefully developed and validated, assuring that the multimodal action performed by the user is actually accomplished by the application. So several frameworks is dedicated to the multimodal interactions, such as [6] and [13].

Here, we focus on a particular multimodal fusion approach: the ICARE approach describing in [3]. ICARE is a component-based approach that is independent of the modalities. So it can be reused without modification in many multimodal applications. Applications described in [2] are made with ICARE components, such as one multimodal user identification system and a prototype of an augmented reality system, allowing to manipulate numeric notes in a real world.

As any other multimodal frameworks, ICARE must ensure a minimum level of quality of its fusion mechanism. This need is increased by the reuse of the ICARE composition components which implements its fusion mechanism. That's why we propose

* Many thanks to Laurence Nigay responsible of the INTUITION project for providing the fusion mechanism-ICARE components.

a validation that targets the composition components of the platform. So our objective is not to test traditional user interfaces, but to validate the way of blending modalities.

The validation is based on testing because we wanted the approach to be applicable in practice. The test architecture is composed of a tool called Tobias [10] that generates a large number of test cases from a scenario, JUnit [8] to execute these test cases and JML assertions (Java Modelling Language [9, 4, 7]) to produce automatically the verdict of test cases. The major interest of this architecture is that it fully supports the testing process: it allows testers to generate many test cases that are automatically evaluated by an oracle. The test case generation proposed by Tobias is an important point of the architecture as it can easily produce a large number of modality combinations with controlled characteristics. So we expect that ICARE can be efficiently tested.

This paper presents the ICARE platform (Sect. 2). It then details the testing infrastructure used for its validation (Sect. 3), focuses on the test methodology (Sect. 4) and reports on the test results (Sect. 5) before drawing the conclusions of the experiment (Sect. 6).

## 2 ICARE platform

### 2.1 Presentation

ICARE is the contraction of Interaction CARE (Complementarity Assignment, Redundancy and Equivalence). It is a component-based approach that allows the easy and fast development of multimodal interfaces with assembled components. ICARE framework is based on a conceptual component model that describes the several software components. A few ICARE components are called "elementary components". Two types of elementary components are defined: "device components" (DC) and "interaction language components" (ILC). A modality is the coupling of a device and an interaction language [13]. For example, in Fig. 1, a speech modality is defined by assembly of one microphone (DC) and the "speech commands" (ILC).

Other ICARE components are called "composition components". They describe combined usages of modalities. The ICARE composition components suggest various fusion mechanisms based on the CARE properties [14]: Complementarity, Assignment, Redundancy and Equivalence. These fusion mechanisms can be applied to any subset of the available modalities. ICARE composition components do not depend on a particular modality and can merge data from two to $n$ modalities. While Equivalence and Assignment express the availability and respective absence of choice among multiple modalities for performing a given task, Complementarity and Redundancy describe relationships between modalities for performing a given task. Thus, three composition components are defined with their own fusion mechanism: the Complementarity one, the Redundancy one and the Redundancy/Equivalence one.

These components have been used in several research applications. For each one, an assembly of components has been set to define which modalities are used and how the modalities are combined. For example, the Figure 1 shows a part of the architecture of a future French military aircraft cockpit prototype. This prototype, called FACET, is described with more details in [3]. Figure 1 shows the ICARE components assembly

in FACET, allowing the pilot to mark a specific point on the ground. For performing a marking command, the pilot has the choice among two modalities. The HOTAS (Hands On Throttle And Stick) modality and the speech modality are functionally equivalent and can be used separately. However, they can also be used in a redundant way, thanks to the Redundancy/Equivalence component. The HOTAS are made of two command joysticks (one for each hand) to pilot the plane and to issue commands. If the pilot presses the HOTAS button or speaks (the voice command <Mark>), one mark command is sent to the rest of the application. If the pilot uses both modalities at the same time, still one mark command is sent. In addition, to detect the target point that the pilot selected in the real world, two modalities are used in a complementary way, thanks to the *Complementarity-1* component. One modality is for the orientation and location of the pilot and the other is for the orientation and location of the plane. Finally, in order to obtain a complete marking command, the command <Mark> must be combined (*Complementarity-2* component) with the target point defined by the pilot.

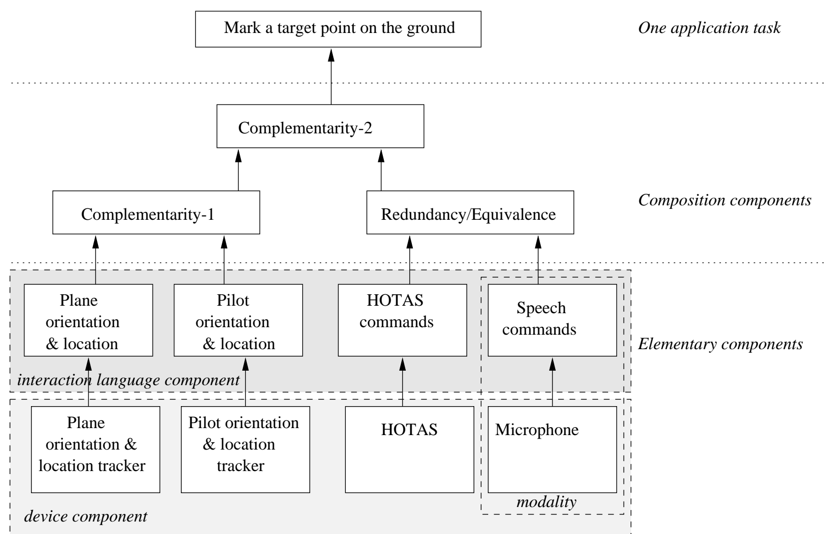


**Fig. 1.** Part of ICARE specification of FACET input interaction

## 2.2 Composition Components

*The Complementarity component* combines all complementary data close in time. For example in Figure 1, orientation and location of both, the pilot and the plane, must be merged to detect the target point on the ground. The mechanism is mostly based on a customizable temporal window, used to trigger the data fusion coming from the modalities. All data coming from the modalities have a confidence factor and a timestamp. To know if orientation and location data are in the same temporal window, the timestamp

of data is used. If the timestamps are too distant, the fusion is not accomplished. The result of the fusion, in addition of the new data generated by the fusion, is a new confidence factor calculated from the merged modalities data. This factor is equal to the average of confidence factors of the merging data.

*The Redundancy component* is used when two or more modalities convey redundant pieces of information that are close in time. In such cases, at least one of the user actions is ignored, because the output is exactly the same. To gain more security, the Redundancy/Equivalence component in figure 1, can be replaced by a Redundancy component. In this case, if only one modality is used (HOTAS or speech), the command <Mark> is not performed, avoiding ambiguous commands and errors. The Redundancy component mechanism seems to be a Complementarity component with the substantial difference that all data coming from the several modalities must be equivalent, corresponding to the same command. Such as Complementarity component, timestamps of data arriving of the modalities are used to detect the redundancy of the command. The new confidence factor is equal to the higher confidence factor among the two modalities data.

*The Redundancy/Equivalence component* is a mix of the two CARE properties Redundancy and Equivalence. It corresponds to the Redundancy component, where the redundancy can be optional. Clearly, if someone has two modalities to perform an action, he can do it in an independent way or in a redundant way. In the redundancy case, only one action is actually performed. As for other composition components, its mechanism is based on a temporal window, but it includes two different strategies: the "eager" and the "lazy" strategies. The "eager" strategy provides an efficient mechanism and the "lazy" strategy provides a safe one. Adopting an "eager" strategy, the component does not wait for further pieces of data to keep propagating data to the following connected ICARE component. Each time a piece of data is sent to another ICARE component, the component keeps its track. It starts a customizable timer in order to detect the redundant pieces of data that may be received later. The advantage of this approach is to give to the user an immediate feedback. The drawback is that the piece of data propagated is the first one received by the component and may not have the highest confidence factor. Opposed to the "eager" strategy, the "lazy" strategy waits until the end of the timer to propagate the piece of data. The advantage of this strategy is to guarantee the propagation of the data containing the highest confidence factor.

## 3 Test Architecture

Although ICARE is only a research prototype, a minimum level of software quality is required to reuse it in the various projects of the our team. Therefore, an effort was initiated in june 2004, in order to check the fusion mechanism of the three ICARE composition components.

As ICARE is coded in Java, we looked for a light validation architecture based on Java technologies. We chose a testing approach because it is the easiest validation technique to carry out. We wanted a test platform that would generate many cases of modality combination and automatically check whether the components react correctly.

### 3.1 Automated oracle

An important topic in software testing is to decide on the success or failure of a given test. This is known as the "oracle problem". In the simplest form of test, it is the test engineer's responsibility to look at the test results and decide on their success/failure (human oracle). This approach requires a lot of effort from the test engineer and does not favour the automation of the test process. Often, the judgement of the test engineer is recorded, so that a replay of the same test can reuse this judgement, provided the output of the program under test is deterministic. In the test of the ICARE platform, it was impossible to use a human oracle:

- The behaviour of the platform is non-deterministic, due to the use of multi-threading in the implementation. It is thus impossible to reuse the results of a test when replaying it.
- The success/failure of the test cannot be decided on what is directly observed by the test engineer. Subtle timing properties must be obeyed that can only be observed by instrumenting the code.
- We intended to play a large number of tests (several thousands), which would require too much interaction for the human oracle.

In this project, the oracle is provided by an executable specification, written in JML (Java Modeling Language [9, 4, 7]). A JML specification is made up of assertions (invariants, history constraints, pre- and post-conditions) which express the properties of the classes and constrain their behaviour. JML assertions appear as comments of the java program. Their syntax is the syntax of java, augmented with several keywords. The JML compiler instruments the code of the program under test, so that assertions are evaluated before and after the execution of each method. As a result, an automated oracle is provided which raises an exception as soon as the behaviour of the code differs from the specified one.

### 3.2 Test execution infrastructure

Fig. 2 shows the various elements of our test execution infrastructure. The goal of the test is to validate composition components, like Complementarity or Redundancy/Equivalence in Fig. 1. These components take as input a series of events, generated by the elementary components, and turn them into higher level events (composed events) which are sent to the application. The test infrastructure embeds each ICARE composition component under test with two classes:

- **an event generator** plays the role of the various modalities corresponding to the elementary components of Fig. 1. The event generator simplifies significantly the test effort, because it decouples the test activity from the concrete aspects of the modalities (e.g. voice and gesture recognition). The test of the elementary components which turn these physical phenomena into computer events, is out of the scope of the validation of the ICARE platform.
- **an observer** collects the events produced by the ICARE composition component.

Both classes are under the control of the popular unit testing tool JUnit [8], which automates the execution of a test suite given as input.

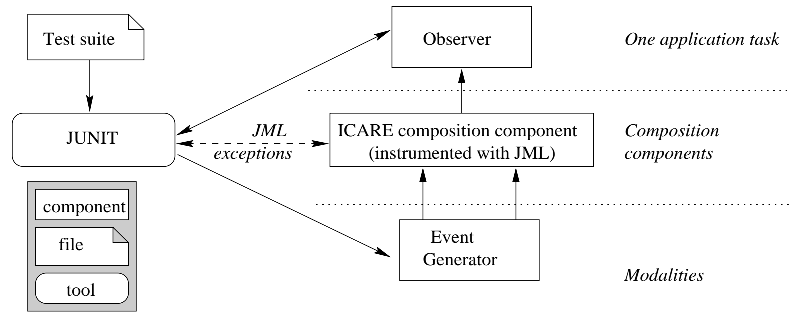


**Fig. 2.** The test execution infrastructure

### 3.3 Finding errors with this approach

The main purpose of the validation work was to find some errors in the code. To be more precise, we tried to find some inconsistencies between the code and the JML assertions. When an error is reported, it can correspond to an error in either the code or in the JML specification. Human analysis is necessary to give the right diagnostic.

Test execution may also lead to detect java run-time errors, e.g. when the Java Virtual Machine runs out of memory. Once again, this may correspond to an error in the java code, or can be the result of the evaluation of the JML assertions. Here again, human diagnostic is needed.

### 3.4 Test suite generation

The test execution infrastructure requires a test suite. In this project, the test suite was generated using the Tobias tool [10]. Tobias is a combinatorial testing tool which starts from an abstract test scenario and unfolds it into a large number of test cases. These test cases, named "abstract test cases", are independent of a specific target technology. Tobias then supports the translation of these abstract test cases into an input file for JUnit (concrete java test cases). Tobias was used to produce many ways to fusion modalities among which some were identified to be particularly important to validate.

## 4 Test description methodology

### 4.1 Approach

As said previously, we applied a combinatorial testing approach to produce a lot of tests. The idea is to (1) identify properties to be validated, (2) express interesting scenarios allowing to observe behaviours with respect to these properties and (3) use Tobias to unfold those scenarios into executable test cases. For example, one can identify that "the Complementarity component combines all complementary data close in time". The time distance has to be defined by the application designer when using the composition components. As said previously, data coming from the modalities have a confidence factor and a timestamp. Thus, the following two points have to be checked:

1. if the timestamps of the data to be merged are too distant, a new data is not produced;
2. if the timestamps of the data to be merged are in the same temporal window, a new data is produced and has a confidence factor equal to the average of confidence factors of the merging data.

One way to do so is to produce tests which first initialise the test infrastructure (i.e. ICARE component, event generator and the observer) This is called “test preambule”. Then, the event generator should send several events with different timestamps and confidence factors (“test body”). Finally, the ICARE component should be stopped (“test postambule”).

This can be expressed as an abstract scenario in Tobias. In the scenario $S_c$ (Fig. 3), the preambule is composed of the five first events

$(G.i; C1.3c; C1.d50; G.st; C1.startComplementatrity())$

and the postambule is composed of the last event ($C.end$). The scenario body consists in 6 *sendComp()* events. The *sendComp()* event is produced by a modality. The majors parameters are $p$ the communication port and $f$ the confidence factor. The scenario is unfolded into 2*6*1*6*2*6 = 864 executable tests. The unfolding operation in Tobias takes only few seconds.

$$S_c = G.i; C1.3c; C1.d50; G.st; C1.startComplementatrity(); \\ G.s1; G.sn; G.s2; G.sn; G.s3; G.sn; C.end;$$

*with*

$$\begin{cases} i = \{G.setEvent(t) | t \in \{1\}\} \\ 3c = \{CC.setNbOfComponents(v) | v \in \{3\}\} \\ d50 = \{CC.setDeltaT(v) | v \in \{50\}\} \\ st = \{setTrace(c) | c \in C1\} \\ s1 = \{sendComp(c, p, d, f, b) | c \in \{C1\}, p \in \{1\}, d \in \{0, 20\}, f \in \{66\}, b \in \{false\}\} \\ s2 = \{sendComp(c, p, d, f, b) | c \in \{C1\}, p \in \{2\}, d \in \{0\}, f \in \{66\}, b \in \{false\}\} \\ s3 = \{sendComp(c, p, d, f, b) | c \in \{C1\}, p \in \{3\}, d \in \{0, 20\}, f \in \{66\}, b \in \{false\}\} \\ sn = \{sendComp(c, p, d, f, b) | c \in \{C1\}, p \in \{1, 2, 3\}, d \in \{0, 20\}, f \in \{33\}, b \in \{false\}\} \\ startComplementatrity() \textit{ is a method with parameter of Complementarity component} \end{cases}$$

**Fig. 3.** One abstract scenario for Tobias

### 4.2 Testing strategy

To validate the 3 components, we produced 19 test schemas, which were unfolded into approximately 6000 test cases. The schemas were designed to test the components in three different ways.

First, tests were designed to check the behavior of the components in normal situations. These tests consists in sending a sequence of various events which are differentiated by their data, their delay i.e. the difference between their initial time and the current time, their confidence factor and the port of the component where they are sent

to. Other parameters of these tests were the delay between two event sendings and the duration *deltaT* that determines if an event is too old to be considered (see schema $S_c$).

The second part of our strategy consists in creating boundary cases that require an intensive solicitation of the component. Some of these tests initialize the components under test with a large number of input ports. Other tests sends a lot of events without delay in between.

Finally some tests aim at considering specific aspects of the ICARE components, in particular their configuration.

We applied this strategy to the three components (Complementarity, Redundancy, Redundancy/Equivalence). In fact, the tests of the Complementarity and Redundancy components were similar as their behaviour is very close to each other: they have to choose a group of input events to determine the information to be propagated. The Redundancy/Equivalence component is different because it has to handle temporal windows. So it was tested with specific test schemas.

## 5 Results of the experiment

### 5.1 Test results

The results of the tests of each ICARE composition component can be summarized into three tables corresponding to the three test strategies. Each table contains four columns: the first column presents the test, the second one the number of test cases produced for this test, the third one the number of errors, the last one the error(s) found. We consider as an error anything that makes the test fail. So it can be a JML/Junit failures or a Java error. These errors are more precisely explained in the comments of the tables.

**Complementarity and Redundancy components** Testing the Complementarity and the Redundancy components gives the same results. It permits to find errors of different types. There are programming bugs such as an event lost (Table 1(b)) or the propagation of too old events (Tables 1(a) and 1(c)).

There is one error resulting from some differences between the specifications and the program. The Redundancy component must check if the input events contain similar data: its implementation checks that the data of input events have the same size whereas the specification requires the same data. The code of the component can be considered as correct if it is assumed that the event-sending device will always send consistent data.

Finally we found an error in the specifications. The specifications state that the number of input ports of a component is unlimited. But this runs the Java Virtual Machine out of memory (Tables 1(c)). So, the specifications and the code of the components should consider the practical constraints and limit the number of input ports.

**Redundancy/Equivalence component** The Redundancy/Equivalence component is implemented differently from the other two components. But it propagates events like the Complementarity and the Redundancy components. So as the other components, it has problems with the content of input events and the modification of parameters when it is running (Table 2(c)).

**Table 1.** Tests for Complementarity and the Redundancy components

| Tests | nb TC | nb E | Error Description |
|---|---|---|---|
| (a) Normal situations | | | |
| sending 6 events to the component | 864 | 1 | propagated events are too old |
| sending 50 events in a multithreaded environment | 784 | 10 | propagated events are too old |
| (b) Boundary tests | | | |
| creating a large number of input ports | 8 | 1 | Java error (no more memory available) |
| sending 6 events to the component without delay | 864 | 10 | propagated events are too old |
| sending 32 events in a multithreaded environment at a high rate | 1 | 1 | one event lost |
| (c) Specific situations | | | |
| changing the configuration of the component while it receives events | 196 | 10 | propagated events are too old |
| initializing the component with various values | 12 | 1 | propagated events are too old |

*nb TC : number of Test Cases for each component*
*nb E : number of error (incorrect behaviour observed during test execution*

Its main difference is the use of threads in order to handle data concurrently. This is the source of several errors found in the component implementation (Tables 2(a) and 2(c)) .

The processing of the component requires concurrent access to the data structure. This brings execution errors (Java errors in Table 2(a)) and an abnormal behavior of the component in lazy strategy. In some cases, calculations are made using accurate information on the state of the data structure, but the result of the calculations is wrong because the state has changed since the event arrival. In other cases, calculations are made using outdated information causing a lack of internal consistency and thus an incorrect behavior.

The eager and lazy strategies are based on the concept of temporal window. Even if boundary tests with only one temporal window ( Table 2(c)) do not bring error, many test failures come from temporal windows (Tables 2(a)). In some cases, an event is not processed at the end of its temporal window. In other cases, delayed events i.e. events arriving into the component some time after their creation are not processed as events arriving at their creation time: these events do not belong to the right temporal window. Moreover the component do not remove events which has a delay longer than the duration of the temporal window.

The Redundancy/Equivalence component being complex, its specification also contains errors. They were found when the component has a correct behavior that the specifications consider incorrect. These cases are reported as “incorrect specification” in the table below.

**Table 2.** Tests for Redundancy/Equivalence component

| Tests | nb TC | nb E | Error Description |
|---|---|---|---|
| (a) Normal situations | | | |
| sending 4 events in the eager strategy | 192 | 30 | 10 wrong behaviors<br>5 to 36 incorrect specification |
| sending 4 events in the lazy strategy | 192 | 20 | 13 wrong behaviors<br>5 incorrect specification<br>2 Java errors |
| (b) Boundary tests | | | |
| sending 5 events during one temporal window in the eager strategy | 64 | 0 | - |
| sending 5 events during one temporal window in the lazy strategy | 64 | 0 | - |
| (c) Specific situations | | | |
| changing the configuration of the component while it receives events | 196 | 71 | 66 wrong behaviors<br>5 incorrect specification |

### 5.2 Advantages of the approach

The test infrastructure chosen is easy to manipulate. In approximately one week, the tester has produced his first tests without all the JML properties. The main difficulty was to write correct JML properties that can be executed. The language offers a large set of constructions. But some construction combination are not executable.

As expected, we fully benefit from the generation of test cases by Tobias. With test schemas, Tobias allows the tester to easily combine in many different ways several modalities. These schemas produced expected test cases, but also unthought test cases that reveals errors. So the combination approach of Tobias is appropriate for multimodality.

One of the most difficult points in testing components is the non-deterministic test execution. So adding JML assertions is a good way to obtain an automatic oracle. But assertions sometimes need access to variables or data that are not directly visible in the program. They require to introduce new piece of code to make the program automatically testable.

### 5.3 Limits of the approach

In theory, each test case should produce a reproducible behavior. However the way components are conceived makes it difficult. In particular, the component behavior is mainly based on the timing of events. But this timing can vary from an execution to another one. So the results of a given test case can change over time. That is why the numbers of failures given in the result tables must be considered as the result approximations of several executions.

The time variations can also cause problems to give a correct test result for the Redundancy/Equivalence component. Because of the use of threads in its structure, some

properties are not verified at execution time even if the component behaves correctly. The shift between the results and the actual behavior occurs in three cases:

- when the temporal window is too small compared to the processing time of an event;
- when the size of the temporal window is comparable to the time between two event arrival;
- when an event delay is greater or equal than the size of the temporal window.

In these cases, the component is likely to be processing an event while the temporal window is closing, making the JML specifications checking the existence of a removed event. To avoid these problematic situations, we chose a "long" temporal window.

Finally failures can come from the specifications. In particular, several failures of the Complementarity and the Redundancy components are related to too old events. The JML processing time can make events become too long. So it is not possible to determine if the failures come from JML or from a too long processing in the components.

## 6 Conclusion

Testing the ICARE composition components has revealed several errors. As expected it permitted to find errors in the fusion mechanism. But it also reveals some cases of modalities fusion that have not been anticipated. So ICARE has really been improved by the experiment which can be considered as a real progress.

The testing tools used has shown the interests of an automatic oracle and of the combinatory test generation for multimodality. Instrumenting the component code by adding JML properties is a light solution to produce an automatic oracle. But it also consumes resources and creates errors by slowing down the program. One way to avoid these problems is to lighten the instrumentation when a problem is detected. For instance, the properties that are necessary for a given test could be commented out. Then the test is played again to see if the problem comes from instrumentation. On the other hand, instrumentation can make some tests succeed by slowing down the execution. In these cases, it is difficult to know that the program is incorrect as it has bad results when it will not be possible to detect them.

In this experiment, we succeed in avoiding the traditional graphical user interface testing pitfalls [11] of automatic test case generation and test oracles. But this is partly due to the fact that we test only components and not a complete interface. So an interesting perspective is to use the same testing infrastructure to validate a whole multimodal application. Good candidates for validation are those that are built with ICARE. First it would be another way of testing the ICARE composition components. Secondly we could use JML to check some ergonomic properties. For instance, we could verify that at the end of the operation that marks a target point on the ground, the mark is really displayed. Of course we could not check that the user sees it, but we could guarantee that the application has a correct behaviour and displays the right information. It is clear that testing real applications would be much more difficult than testing components, in particular because of the multi-threading. This could require to change some parts of the testing infrastructure, particularly JUnit that executes tests.

As a conclusion, the experiment related in this paper had a very positive result: the ICARE framework is more robust than it used to be and the extensive test campaign has increased our confidence in its quality. We believe that several test techniques are now mature enough and sufficiently easy to use, to be applied to other fusion mechanisms, and that this will help master the complex development of multimodal user interfaces.

## References


1. R. A. Bolt. "Put-that-there": Voice and gesture at the graphics interface. In *SIGGRAPH'80*, pages 262–270, 1980.
2. J. Bouchet and L. Nigay. ICARE: A Component-Based Approach for the Design and Development of Multimodal Interfaces. In *Extended Abstracts of CHI'04*, pages 1325–1328, Vienna, Austria, 2004.
3. J. Bouchet, L. Nigay, and T. Ganille. ICARE Software Components for Rapidly Developing Multimodal Interfaces. In *ICMI'04*, pages 251–258, State College, PA, USA, 2004.
4. L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.
5. P. R. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow. QuickSet: Multimodal interaction for distributed applications. In E. Glinert, editor, *Proceedings of the Fifth ACM International Multmedia Conference*, pages 1325–1328. ACM Press, New York, 1997.
6. F. Flippo, A. Krebs, and I. Marsic. A Framework for Rapid Development of Multimodal Interfaces. In *ICMI'03*, pages 109–116, 2003.
7. The Java Modeling Language (JML) Home Page. http://www.cs.iastate.edu/ leavens/JML.html.
8. JUnit. http://www.junit.org.
9. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
10. Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering TOBIAS combinatorial test suites. In *Fundamental Approaches to Software Engineering (FASE'04)*, volume (to appear) of *LNCS*, Barcelona, Spain, 2004. Springer.
11. A. Memon. GUI Testing: Pitfalls and Process. *Software technologies*, pages 87–88, 2002.
12. L. Nardelli, M. Orlandi, and D. Falavigna. A Multi-Modal Architecture for Cellular Phones. In *ICMI 2004*, pages 323–324, State College, PA, USA, 2004.
13. L. Nigay and J. Coutaz. A Generic Platform for Addressing the Multimodal Challenge. In *CHI'95*, pages 98–105, 1995.
14. L. Nigay and J. Coutaz. The CARE Properties and Their Impact on Software Design. In *Intelligence and Multimodality in Multimedia Interfaces*, 1997.
15. S. Oviatt and al. Designing the user interface for multimodal speech and gesture applications: State-of-the-art systems and research directions. *HCI*, 15-4:263–322, 2000.