Demonstrating and Testing the BML Compliance of BML Realizers

Herwin van Welbergen[1], Yuyu Xu[2], Marcus Thiebaux[2], Wei-Wen Feng[2], Jingqiao Fu[2], Dennis Reidsma[1], and Ari Shapiro[2]

[1] Human Media Interaction, University of Twente
{h.vanwelbergen@utwente.nl,d.reidsma}@utwente.nl
[2] Institute for Creative Technologies, University of Southern California
shapiro@ict.usc.edu

Abstract. BML realizers are complex software modules that implement a standardized interface –the BML specification language– to steer the behavior of a virtual human. We aim to promote and test the compliance of realizers that implement this interface. To this end we contribute a corpus of example BML scripts and a tool called RealizerTester that can be used to formally test and maintain adherence of realizers to the BML standard. The standardized interface of realizers allowed us to implement RealizerTester as an automatic testing framework that can test *any* realizer. RealizerTester can 1) help in maintaining the stability and extensibility that is crucial for realizers and 2) contribute to the formalization of the emerging BML standard, both by providing test scripts and a formal description of their constraints and by identifying and resolving execution inconsistencies between realizers. We illustrate the testing practices used in the development of two realizers and demonstrate how RealizerTester is integrated with these practices. The scripts in the example corpus were executed on both realizers. This resulted in a video corpus that demonstrates the semantic equivalences and differences in execution of BML scripts by the two realizers.

1 Introduction

The SAIBA framework [5,9] has standardized the architecture of virtual human applications with the aim of making reuse of their software components possible. The SAIBA framework proposes a modular 'planning pipeline' for real-time multimodal motor behavior of virtual humans, with standardized interfaces (using representation languages) between the modules in the pipeline. One of the components in this pipeline is the realizer. A realizer provides an interface to steer the motor behavior of a virtual human: a description of behavior in the Behavior Markup Language (BML) goes 'in', feedback comes 'out'.

Several realizers have been implemented [8,6,4,2,10]. If SAIBA's goal of software reuse is achieved, it will be possible to use such realizers interchangeably with the same BML input. We are interested in measuring and promoting this compatibility between realizers and to provide tools to formally test and maintain adherence to BML standard. To this end, we provide a growing test set of

H. Högni Vilhjálmsson et al. (Eds.): IVA 2011, LNAI 6895, pp. 269–281, 2011.

BML test cases, a corpus of BML scripts and video material of their realization in different (so far, two) realizers: SmartBody [8] and Elckerlyc [10].

By directly comparing BML realizers, we can better determine changes to the BML specification that are necessary due to overly narrow or broad specifications. Overly broad specifications can be detected when realizers provide BML compliant, but semantically very different results. Overly narrow specifications indicate that a specification is not expressive enough, they can be detected when several Realizers implement the same (semantic) functionality, yet require Realizer specific proprietary BML extensions to implement (part of) this functionality.

Since each realizer necessarily implements the same interface, an automatic testing framework can be designed that tests the adherence to the BML/feedback semantics for *any* realizer. We contribute our testing framework RealizerTester[1], which provides exactly this functionality.

2 On BML Versions and Script Creation

Currently, there are two version of the BML specification: a first draft specified after the BML workshop in Vienna in November 2006 (the Vienna draft) and the current draft of BML version 1.0 (draft 1.0). Draft 1.0 is not backward compatible with the Vienna draft. Currently Elckerlyc implements draft 1.0, and SmartBody implements the Vienna draft. It is likely that new versions of BML will be developed[2] and that not all realizers will adapt to these new versions at the same pace. However, test scripts can be constructed that are semantically equivalent (that is, execute behavior that adheres to the same form and timing constraints) for most, if not all, BML behaviors in different versions of BML. This implies that the same test case (albeit not test script) can be used to test realizers that implement different versions of BML.

We aim to construct a test set that contains such semantically equivalent test cases for all BML versions. These tests provides a 'safety net' for migrating a realizer from a previous BML version to the next; the tests that worked for a realizer in the previous version should not break in their updated syntax in a next version of BML.

The process of converting the old tests to a new BML version also helps in the definition of the standard. It can highlight certain cases in which expressivity is lost where this might not be intended. That is: if something can be expressed in a previous version of BML which we cannot express in the new version of BML and this loss of expressivity was not explicitly intended in the new BML version, then their might be something 'wrong' in the definition of the new version.

Most of our current test scripts were originally designed for draft 1.0 and later converted to equivalent Vienna draft scripts. During this conversion process, we have encountered several cases that demonstrate the enhanced expressivity of the newer draft 1.0. For example:

[1] RealizerTester is released under the MIT license at `http://sourceforge.net/projects/realizertester/`

[2] The BML workshop at this IVA aims to finalize BML 1.0.

- In the Vienna draft it was impossible to specify a realizer-independent posture behavior; draft 1.0 provides the specification of some default lexicalized postures.
- Draft 1.0 provides the specification of a modality (e.g. eyes, neck, torso) in gaze behaviors. The Vienna draft does not allow this. Therefore, SmartBody currently needs to use a custom extension to specify the modality of its gaze behavior.

A small set of scripts was converted from a Vienna draft specification to a draft 1.0 specification. So far we have not encountered any cases in which expressivity of the Vienna draft was lost in draft 1.0.

3 A Corpus of Test Cases and Videos

We provide a growing corpus of BML scripts for the purpose of visual comparison between the execution of a BML script by the different realizers. A matching video corpus illustrates how these scripts are executed in both SmartBody and Elckerlyc. The corpus of BML scripts (and matching video) provides examples of both short monologues and of the execution of isolated behaviors. The companion video and Fig. 1 compare the execution of some of these scripts in SmartBody and Elckerlyc. Here we discuss some preliminary observations on the comparison of such videos.

Script are included for most types of BML behavior. In the scripts used so far, the `speech`, `gaze`, `head` and pointing (part of `gesture`) behaviors look similar in Elckerlyc and SmartBody. A comparison of `face` behaviors, specified through Ekmans FACS [3] gives mixed results, as illustrated in the video. The `posture` behavior could not be compared in a meaningful manner, because the Vienna BML draft does not provide a realizer independent way to specify posture. The `locomotion` behavior is currently not implemented in Elckerlyc and is therefore omitted from the visual comparison corpus for now.

When designing short monologues for the visual comparison corpus, it became clear that existing demonstration scripts of both SmartBody and Elckerlyc rely heavily upon custom behavior elements. One reason for this is the lack of expressivity of the BML standard and specifically the lack of expressivity in the specification of iconic and metaphoric gestures. We recommend, at the very least, to extend the lexicalized set of gestures that can be specified in BML to include more non domain specific gestures. The set of gestures that is already implemented through extensions by current realizers could serve as an inspiration for this. Another reason for the use of custom BML elements is that so far there was no real need for BML compliance of realizers. Some BML elements that are currently implemented using one or more custom BML extensions in Elckerlyc and SmartBody could be implemented in standard BML. This testing and comparison corpus building effort serves as a driving force for this. Already some new core BML behaviors were implemented in both realizers to achieve better BML compliance.

We have created a test corpus containing 19 test scripts in BML draft 1.0, and corresponding test cases that check, e.g., the adherence to the time constraints specified in the scripts. We are currently in the process of converting the test scripts to the Vienna BML draft.

4 Automatic Software Testing of Realizers

Realizers are complex software components. They often form the backbone of several virtual human applications of a research group. Therefore, the stability and extensibility of realizers is crucial. So far, the testing of most of these realizers was limited to *manual*, time consuming inspections of the execution of a selected set of BML scripts [1]. *Automatic* testing can be used to detect errors in realizers and provide a 'safety net' that can, to some extent, ensure that extensions or design cleanups did not introduce a failure in existing functionality. Since the automatic tests do not require manual intervention, they can be run often which ensures that errors are detected early, which makes it easier to fix them.

RealizerTester provides an automatic testing framework for Realizers. We illustrate the use of RealizerTester by describing how it was integrated in the software development process of the Elckerlyc and SmartBody realizers and discuss how it can contribute to the emerging BML standard.

A Behavior Planner communicates with a realizer by sending BML blocks with intended behavior to it and by capturing the feedback provided by the realizer. A BML block defines the form and relative timing (using constraints on its sync points, see also Fig.2) of the behavior that a realizer should display on the embodiment of a virtual human. The realizer is expected to provide the Behavior Planner with feedback on the current state of the BML blocks it is executing: it notifies the Behavior Planner of the start and stop of each BML block (performance start/stop feedback) and on the passing of sync points for each behavior in the block (sync-point progress feedback). Execution failures are sent using warning and exception feedback.

RealizerTester acts as a Behavior Planner: it sends BML blocks to the Realizer Under Test (RUT)[3] and verifies if the feedback received from the RUT satisfies the assertions implied by the BML blocks. This allows automatic testing of the following properties:

1. *Message Flow and Behavior Execution*: RealizerTester can verify if the performance start/stop of each BML block and sync-point progress feedback messages of each behavior was received in the correct order and only once. This implicitly provides some information on whether or not the behaviors were actually executed.
2. *Time Constraint Adherence*: a BML block defines several time constraints upon its behaviors. It can require that a sync point in one behavior occurs simultaneously with a sync point in another behavior, or that a certain sync points should occur before or after another one. These constraints can be tested by inspecting the sync-point progress feedback.

[3] After System Under Test used in [7].

Fig. 1. Execution of some BML behaviors in Elckerlyc (left) and SmartBody(right). From top to bottom: AU 1 (inner eyebrow raise), AU 6 (cheek raiser and lid compressor), gaze, point.

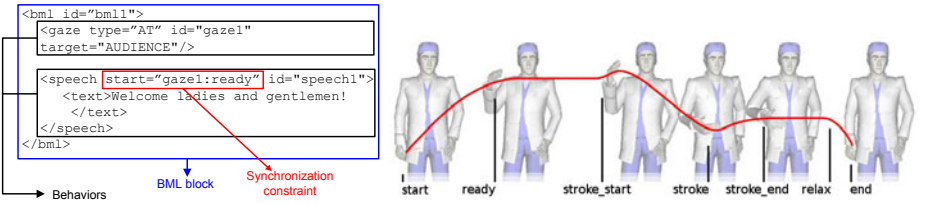


Fig. 2. An example BML script and the sync points of a BML behavior

3. *Error Handling*: The error handling of a realizer can be tested by inspecting its warning and exception feedback. For example, RealizerTester can send BML blocks to the realizer that are invalid or impossible to schedule and then check if the realizer generated the appropriate exception feedback.

4.1 Test Architecture

Each automatic test consists of four phases [7]:

1. *Fixture setup*: The Fixture contains the RUT and everything it depends on to run. During Fixture set up, the RUT is created and put in a state suitable for testing. The necessary functionality to keep track of the feedback sent by the RUT is also hooked up.
2. *Exercise the RUT*: Send BML block(s) to the RUT.
3. *Result verification*: Verify the feedback received from the RUT.
4. *Fixture teardown*: Clean up the fixture.

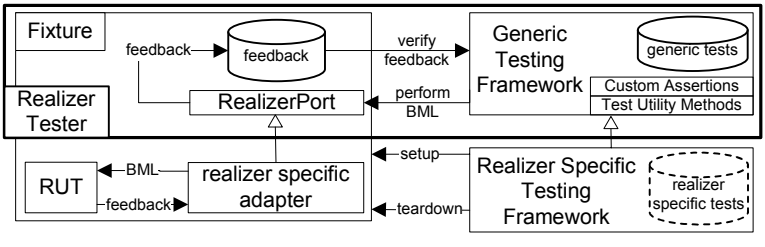


Fig. 3. Testing Architecture

Phase 1 and 4 are realizer specific, phase 2 and 3 are generic. The Generic (realizer independent) Testing Framework contains a set of tests and is responsible for exercising these tests and verifying their results. The Generic Testing Framework exercises the RUT by communicating with it through a RealizerPort. The RealizerPort is a minimal interface for a realizer.

A Realizer Specific Testing Framework is responsible for setting up and tearing down the Fixture before and after each test case. During the setup phase, this framework creates a realizer specific implementation of the RealizerPort

and connects it to the RUT. RealizerTester is implemented using the JUnit[4] unit testing framework. Since the Fixture setup and teardown is the same for each test, they are implemented using setup and teardown functions that are called automatically before and after each test respectively.[5] Fig. 3 shows our architecture setup.

4.2 Authoring Test Cases

Test cases are typically set up as follows:

1. Send one or more BML blocks to the RUT, capture all feedback.
2. Wait until the RUT has finished executing all blocks.
3. Verify some assertions on the received feedback.

RealizerTester provides several Test Utility Methods and Custom Assertions [7] to help a test author with this. In the test setup phase, the RUT is coupled to a feedback handler that stores all feedback messages. Most of the Test Utility Methods and Custom Assertions act upon this feedback messages storage. The Custom Assertions in RealizerTester verify various commonly required assertions on the received feedback and provide meaningful error messages if these assertions fail.

Fig. 4 shows an example test case consisting of a BML block (top) and a test function executing the BML block and verifying the assertions implied by the block (bottom). Note that the test case is fully specified using Custom Assertions and Test Utility Methods, which make it very readable.

Many realizers support custom BML behavior elements. Such elements can be tested using test cases in the realizer Specific Framework. The Custom Assertions and Test Utility Methods described above can help in the creation of such test cases.

5 Employing RealizerTester in Elckerlyc

The BML specification is an emerging standard, and at the moment of writing, there are no realizers that fully implement the BML/feedback interface proposed by the SAIBA initiative. Elckerlyc [10] implements several (but not all) BML behaviors and supports BML feedback. This made it a good first test-candidate for RealizerTester. Here we describe our experiences with the integration of RealizerTester in Elckerlyc's software development process.

We have implemented an Elckerlyc Specific Testing Framework that sets up a Fixture that uses Elckerlyc as its RUT. Elckerlyc is tested using the 19 test cases provided by RealizerTester. An additional 12 test cases were implemented to test BML behaviors that are specific to Elckerlyc.

[4] `http://www.junit.org/`

[5] Meszaros [7] calls this Implicit Setup and Teardown, functionality for this is available in JUnit.

```
<bml id="bml1">
    <speech id="speech1" start="6">
        <text>Hey punk <sync id="s1" />what do ya want?</text>
    </speech>
    <head id="nod1" action="ROTATION" rotation="X" start="speech1:s1"/>
</bml>

@Test public void testSpeechNodTimedToSync() {
   realizerPort.performBML(readTestFile("testspeech_nodtimedtosync.xml"));
   waitForBMLEndFeedback("bml1");
   assertSyncsInOrder("bml1", "speech1", "start", "ready", "stroke_start",
       "stroke", "s1", "stroke_end", "relax", "end");
   assertAllBMLSyncsInBMLOrder("bml1", "nod1");
   assertBlockStartAndStopFeedbacks("bml1");
   assertRelativeSyncTime("bml1", "speech1", "start", 6);
   assertLinkedSyncs("bml1", "speech1", "s1", "bml1", "nod1", "start");
   assertNoExceptions();
   assertNoWarnings();
}
```

Fig. 4. An example test case. A BML block (top) is sent to a RUT and the test function awaits the end feedback for the block (using the `waitForBMLEndFeedback` Test Utility Method). It then verifies the correctness of the execution using various assertions. The Custom Assertions `assertSyncsInOrder`, `assertAllBMLSyncsInBMLOrder`, and `assertBlockStartAndStopFeedbacks` verify the message flow and behavior execution. They validate respectively that feedback on the syncs points of the `speech1` and `nod1` behavior was received in the correct order, and that the performance start and stop feedback for the block was received once. The BML block specifies that sync point `speech:start` should occur at relative (to the start of the block) time 6, and that sync point `speech1:syncstart1` should occur at the same time as `nod1:start`. The Custom Assertions `assertRelativeSyncTime` and `assertLinkedSyncs` verify these scheduling constraints. Finally, the Custom Assertions `assertNoExceptions` and `assertNoWarnings` verify that the block was executed without failure.

Automatic testing has proven useful in both finding errors in Elckerlyc and making sure that new functionality did not introduce errors. In some cases it was useful to define test cases as acceptance tests for new functionality *before* it was implemented.[6] One such test highlighted deficiencies in Elckerlyc's BML scheduling algorithm. Passing the test (by an update to the scheduling algorithm that fixed these deficiencies) marked the implementation of a certain software requirement.

Automatic testing is more valuable if it is done as often as possible. However, running all test cases on RealizerTester takes some time (roughly 3 minutes on our test set of 31 tests), which might discourage its frequent use by Elckerlyc's

[6] This is a common practise in the Test Driven Development software development process [7].

developers. We have solved this issue by running the tests automatically on Elckerlyc's continuous integration server[7] whenever a developer commits changes to its source repository. If a test fails, the developer responsible for the test failure is automatically notified. The integration server also keeps track of the test performance over all builds, so it is possible to identify exactly what build introduced an error. RealizerTester helps the Elckerlyc developers in the notification of errors, but it does not directly help in identifying the exact location of errors, since it is testing the realizer as a black box. The use of white box testing at a smaller granularity helps in Elckerlyc's defect localization. To this end, over 1000 unit tests (typically testing one class) and mid-range tests (testing groups of classes working together) are employed to test Elckerlyc. The unit tests run fast (in under 10 seconds) so developers run them very often to check the health of newly created code. The test cases by RealizerTester are used in Elckerlyc to test how different (unit tested) components work together as a whole, and, if tests fail, as an indication of locations that require more unit testing.

Automatic testing is useful because it can be done as often as possible without cost (e.g. in developer/human tester time). However, we have found that manual inspection is more flexible than the rigid assertion verification employed by automatic tests and that some errors in Elckerlyc can currently only be identified by manual inspection of the behavior of a virtual human. Therefore we recommend regular visual inspection in addition to automatic testing.

Most visualization failures that have occurred in Elckerlyc so far are a result of physical simulation errors, resulting in gross movement errors (e.g. the virtual human falling over, rolling through the scene uncontrollably, showing large 'hitches' in movement etc.). The occurrence of such failures is often dependent on the specific set of movement combinations and the virtual human embodiment used. Because of this, there is often a long time span between the introduction of such failures in the code base and their discovery, which makes the failures hard to repair. SmartBody's testing framework contributes automatic visual regression testing (discussed in the next section). Integrating such automatic visual testing in Elckerlyc would be quite helpful to detect these and other visual regressions in a timely fashion.

6 Testing in SmartBody

SmartBody contributes a test program that provides automatic *visual* regression testing mentioned above. This test program takes screen snapshots at predefined moments in an ongoing simulation (e.g. the execution of a BML script). A baseline of such screen snapshots is saved as input for subsequent test simulations. During a later simulation, another snapshot at the same virtual time in the simulation is taken and then compared pixel by pixel against the baseline image. If the images differ more than a predefined threshold (see below), then the test can be marked as failing and examined manually by a tester. To this end, the

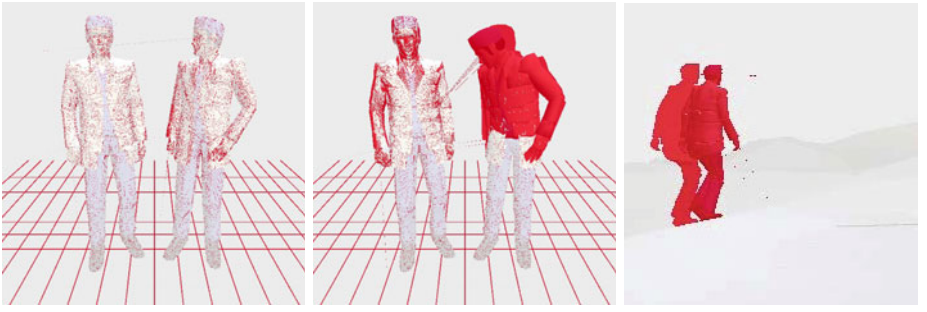
[7] Elckerlyc uses Jenkins (`http://jenkins-ci.org/`)

test program can provide the tester with an image that shows the baseline image overlayed with the differences from the test snapshot. Fig. 5 shows some examples of such difference images. A similar visual difference regression testing method is used in the graphics industry [11].

By comparing the results of all aspects of the simulation via a graphical image, the tester is better able to determine the impact of various changes to the realizer. When implementing this method, it is important to position the camera where it can detect meaningful differences between the images during a test run. For example, to test head nodding, the camera should be positioned close to and to the side of the virtual human's face. Also, randomness during simulations, such as reliance on real-time clocks, needs to be eliminated in order to generate repeatable results. Different platforms and graphics drivers will also tend to produce similar, but not identical results. This problem can often be mitigated by setting a sufficiently high image comparison threshold. Changes in functionality will often change the results of the test images that are desirable. When this happens, the tester would then create a new set of baseline test images based on the new functionality. Authoring such a graphical test thus involves choosing a certain simulation, defining a set of important times to check for image differences, defining a camera position and setting an image comparison threshold (using the default value often suffices).

SmartBody's test program goes beyond the tests performed by RealizerTester by checking if the correct motion is generated, rather than by checking if the Realizer sends the correct signals. However, unlike RealizerTester, SmartBody's testing system is Realizer specific and limited to provide only regression (and not acceptance) testing functionality. This testing method is thus complementary to testing by RealizerTester.

Therefore, RealizerTester is also used to test SmartBody. To this end, SmartBody has been extended to allow the feedback messages required by RealizerTester, and our (draft 1.0) test scripts have been converted to the Vienna draft BML standard used in SmartBody. Allowing SmartBody to be tested with RealizerTester involved creating a SmartBody Specific Testing Framework and a SmartBody adapter of RealizerPort (see also Section 4.1). This is a relatively simple effort, taking roughly one day of programming. The BML standard only specifies the information that should be contained in the feedback, but does not specify the exact form/syntax of feedback. As a result, much of the implementation effort was spend in parsing SmartBody feedback and converting it to a suitable form for RealizerTester. The process of connecting RealizerTester to a new realizer is similar as connecting any behavior planner to a new realizer. This means that behavior planner developers have to implement error prone and somewhat elaborate feedback parsing for each realizer they connect their behavior planner to. We strongly suggest to incorporate a standard syntax (for example in XML) for feedback in the BML standard to alleviate this issue. By testing SmartBody with RealizerTester, some minor implementation issues in SmartBody were discovered. We did not find any interpretation differences in the constraint satisfaction between SmartBody's and Elckerlyc's realization of



(a) (b) (c)

Fig. 5. Fig. 5(a): The simulation run differs from its baseline mostly by subtle differences in the position of the triangular mesh that represents the virtual human's body parts. These differences should not surpass the comparison threshold and should result in a successful test. Fig. 5(b): The character on the right differs from the baseline in the amount of forward lean towards the gaze target. This difference should exceed the comparison threshold and result in a failed test. Fig. 5(c): A test of SmartBody's locomotion system on uneven terrain. The baseline is shown on the right, the results from the new test (with a different parameter configuration for e.g. walking velocity) is shown as a silhouette of the virtual human on the left.

the test scripts. A minor difference in message flow was found. In Elckerlyc, sync-point progress feedback messages are guaranteed to be sent in order (first start, then ready, then stroke_start, then stroke, then stroke_end, then relax, finally end). The performance start messages of a BML block is guaranteed to occur before all sync-point progress feedback messages of the behaviors in the block. The performance stop message of the BML block is guaranteed to occur after all sync-point progress feedback messages are sent. SmartBody does not enforce such a message order; sync-point progress feedback messages and/or performance start/stop feedback messages that occur at the same time are sent in an undefined order.

7 Conclusion and Discussion

We have provided a corpus of BML behaviors and video material of their realization in two realizers and a corpus of BML test cases. We aim to increase the size of these corpora and welcome additions to them, especially from the authors of realizers other than SmartBody and Elckerlyc. The test corpus and the visual comparison movie and script corpus are available online under a creative commons license.[8]

Preliminary inspection of the video corpus shows some expressivity issues in the BML standard, but also shows that several behaviors are executed on the

[8] `http://sourceforge.net/projects/realizertester/`

two different realizers in a semantically equivalent manner. More importantly, the process of creating the corpus created healthy competition between different groups building realizers, each trying to enhance the animation quality of their realizers to out do the other. It also motivated them to move toward more compliance to the BML standard.

The modularity proposed by the SAIBA framework not only allows the reuse of existing realizers with new Behavior Planners, but also reuse of test functionality and test cases for different realizers. The same modularity that allows one to connect a Behavior Planner to any realizer, also allows RealizerTester to test any realizer. RealizerTester and its test cases provides a starting point for a test suite that can test the BML conformance of realizers. Such conformance tests are common for software that interprets XML.[9]

When designing BML test scripts *and* their corresponding test assertions we ran into several cases in which the BML specification lacked detail, was unclear, or was unfinished. For example, the current BML specification does not state whether two posture behaviors (using different body parts) could be active at the same time, if gaze can be a persistent behavior, or how custom sync points in a behavior are to be aligned in relation to its default BML sync points. The process of designing a test set of BML scripts and their corresponding test assertions can significantly contribute to the improvement of the BML standard itself, by highlighting such issues.

Currently each test case consists of an BML script (in a separate XML file) and a test function in RealizerTester. It would be beneficial to merge the test function itself into the BML script, so that new tests can easily be authored without modifying the source code of RealizerTester itself. For many BML scripts (e.g. those that do not deliberately introduce error feedback or change the behavior flow), it should even be possible to automatically generate test assertions directly from the script rather than authoring them by hand.

The BML standard contains several open issues and its interpretation may vary between realizer developers. If adapted by multiple realizers, RealizerTester can contribute to the formalization of the BML standard (by providing BML test scripts and a formal description of their constraints, as expressed in test assertions) and help identify and resolve execution inconsistencies between realizers. A realizer does not need to be fully BML compliant to be tested by RealizerTester; supporting feedback and some BML behaviors is sufficient. We invite the authors and users of realizers to join our realizer testing effort by contributing test cases and hooking up their realizers to RealizerTester.

Acknowledgements. This research has been supported by the GATE project, funded by the Dutch Organization for Scientific Research (NWO).

[9] For example, for Collada (`http://www.khronos.org/collada/adopters/`) or XHTML (`http://www.w3.org/MarkUp/Test/`) interpreters.

References

1. Personal communication with the authors of SmartBody, Greta, EMBR, and RealActor (2010)
2. Čereković, A., Pandžić, I.S.: Multimodal behavior realization for embodied conversational agents. In: Multimedia Tools and Applications, pp. 1–22 (2010)
3. Ekman, P., Friesen, W.: Facial Action Coding System: A Technique for the Measurement of Facial Movement. Consulting Psychologists Press, Palo Alto (1978)
4. Heloir, A., Kipp, M.: Real-time animation of interactive agents: Specification and realization. Applied Artificial Intelligence 24(6), 510–529 (2010)
5. Kopp, S., Krenn, B., Marsella, S., Marshall, A.N., Pelachaud, C., Pirker, H., Thórisson, K.R., Vilhjálmsson, H.H.: Towards a common framework for multimodal generation: The behavior markup language. In: Gratch, J., Young, M., Aylett, R.S., Ballin, D., Olivier, P. (eds.) IVA 2006. LNCS (LNAI), vol. 4133, pp. 205–217. Springer, Heidelberg (2006)
6. Mancini, M., Niewiadomski, R., Bevacqua, E., Pelachaud, C.: Greta: a saiba compliant eca system. In: Troisiéme Workshop sur les Agents Conversationnels Animés (2008)
7. Meszaros, G.: xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, Reading (2007)
8. Thiebaux, M., Marshall, A.N., Marsella, S., Kallmann, M.: Smartbody: Behavior realization for embodied conversational agents. In: Autonomous Agents and Multiagent Systems, pp. 151–158 (2008)
9. Vilhjálmsson, H.H., Cantelmo, N., Cassell, J., E. Chafai, N., Kipp, M., Kopp, S., Mancini, M., Marsella, S.C., Marshall, A.N., Pelachaud, C., Ruttkay, Z., Thórisson, K.R., van Welbergen, H., van der Werf, R.J.: The behavior markup language: Recent developments and challenges. In: Pelachaud, C., Martin, J.-C., André, E., Chollet, G., Karpouzis, K., Pelé, D. (eds.) IVA 2007. LNCS (LNAI), vol. 4722, pp. 99–111. Springer, Heidelberg (2007)
10. van Welbergen, H., Reidsma, D., Ruttkay, Z.M., Zwiers, J.: Elckerlyc: A BML realizer for continuous, multimodal interaction with a virtual human. Journal on Multimodal User Interfaces 3(4), 271–284 (2010)
11. Yee, Y.H., Newman, A.: A perceptual metric for production testing. In: ACM SIGGRAPH Sketches, p. 121. ACM, New York (2004)