Elckerlyc in practice – on the integration of a BML Realizer in real applications

Dennis Reidsma and Herwin van Welbergen*

Human Media Interaction, University of Twente, the Netherlands
d.reidsma@utwente.nl, http://hmi.ewi.utwente.nl

Abstract. Building a complete virtual human application from scratch is a daunting task, and it makes sense to rely on existing platforms for behavior generation. When building such an interactive application, one needs to be able to adapt and extend the capabilities of the virtual human offered by the platform, without having to make invasive modifications to the platform itself. This paper describes how Elckerlyc, a novel platform for controlling a virtual human, offers these possibilities.

Key words: Virtual Humans, Embodied Conversational Agents, Architecture, System Integration, Customization

1 Introduction

Virtual Humans (VHs) are used in many educational and entertainment settings: serious gaming, interactive information kiosks, kinetic and social training, tour guides, storytelling entertainment, tutoring, interactive virtual dancers, entertaining games, motivational coaches, and many more. Building a complete VH from scratch is a daunting task, and it makes sense to rely on existing platforms. However, when one builds a novel interactive VH application, one needs to be able to adapt and extend the capabilities of the VH offered by the platform, without having to make invasive modifications to the platform itself.

The SAIBA framework [1] provides a good starting point for designing interactive VHs. Its emerging Behavior Markup Language (BML) defines a specification of the form and relative timing of the behavior (e.g. speech, facial expression, gesture) that a BML Realizer should display on the embodiment of a VH.

Elckerlyc is a state-of-the-art BML Realizer. Elsewhere, we described its mixed dynamics capabilities, that allow one to combine physics simulation with other types of animation, and its focus on continuous interaction, which allows it to monitor its own performance and allows for last moment modification of behavior plans with respect to content and timing, which makes it very suitable for VH applications requiring high responsiveness to the behavior of the user [2]. Here, we will focus on its role as a component in a larger application.

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

2 Requirements for a modular and extensible realizer

An application that uses a VH as one of its components might have several requirements for the BML Realizer. Specific additional gestures and face expressions might be needed; the application might need to run distributed over several machines; the experimenter might need detailed logs of everything that the VH does; one might want to replace the graphical embodiment of the VH, or its voice; the embodiment of the VH might need to reside in a custom game engine instead of Elckerlyc's default renderer; and one might need to plug in completely new custom behaviors and modalities for a specific usage context.

Developing extensions or alternative configurations of Elckerlyc should be possible without requiring changes to the core Elckerlyc system (that is, extensions should not require recompilation of the Elckerlyc source). After all, if Elckerlyc extensions lead to a modification of Elckerlyc itself, then this would essentially lead to a separate Elckerlyc fork for every application using Elckerlyc. This would make it difficult to share new extensions with the community. Also, once Elckerlyc has been forked to accomodate a new modality engine or behavior type, it becomes difficult to take advantage of improvements in the 'core' Elckerlyc source: they need to be painstakingly merged into the fork.

Below follows a number of extensibility requirements for Elckerlyc, that should be implemented as *non-invasive modifications*: they may entail the implementation of new *run-time libraries*, or the addition of new *resources*; but should not require *compile time* dependencies for Elckerlyc on new code.

- Integration with new renderers, speech synthesizers, physics simulators, ...
- Flexible ways to send BML to the realizer, and to adapt the BML stream with capabilities for filtering and logging.
- Provide a transparent mapping from input (BML behavior elements) to output (control of the VH's embodiment).
- Provide possibilities to add new behavior types or output modalities.
- Provide easy ways to integrate a BML Realizer as a component in an application, independent of variables such as the OS and programming language on which the application is developed.

3 Related Work

Like Elckerlyc, the BML Realizers Smartbody [3], EMBR [4] and Greta [5] were specifically designed for integration with existing renderers, to allow a wide range of behavior types, and/or to facilitate integration in different applications. Elckerlyc additionally contributes a transparent and adjustable mapping from BML to output behaviors (rather than the mostly hardcoded mappings in other realizers), and allows for easy integration of new modalities and embodiments, for example to control robotic embodiments. In this section, we discuss how various requirements were solved for the three realizers mentioned above, and shortly indicate the differences with our solutions. In the next section, we will go deeper into the solutions used in Elckerlyc, also showing how they impact actual use.

Integration with existing renderers Smartbody provides the BoneBus library to connect the Smartbody realizer to a renderer. BoneBus uses UDP to transport (facial) bone positions and rotations from the realizer to the renderer. BoneBus is designed to hide the details of the exact communication protocol used, so that its exact implementation can be changed at a later stage without changing realizers or renderers that use the library. As the data transport protocol is non-trivial and due to change, reimplementing BoneBus in programming languages other than C++ or using the BoneBus interface with other transport mechanisms (TCP/IP, shared memory, etc.) is infeasable. Currently, SmartBody has been integrated with a number of renderers. The output of Greta contains MPEG-4 facial and body action parameters. By using the MPEG-4 standard, Greta can potentially be used with any renderer that supports MPEG-4. However, MPEG-4 –especially for body animation– is not widely supported.

Elckerlyc currently uses the Thrift remote procedure call (RPC) framework [6] to handle its communication with the renderer. Unlike the BoneBus library, this allows us to set up a communication channel that is agnostic to the programming language used on either side and that allows one to configure and change the mode of transport (e.g. TCP/IP, shared memory, pipes).

Available Behavior Types and Extensibility Smartbody use keyframe animation and a fixed set of biologically motivated motion controllers (e.g. for gaze) to achieve facial and body motion. EMBR uses keyframe animation, procedural animation with a fixed set of expressive parameters, autonomous motion (such as eyeblink and balancing), morph targets for facial animation, and controllable shaders (e.g. for blushing). Greta uses procedural body animation with a fixed set of expressivity parameters, and Ekman's action units [7] for facial animation.

Elckerlyc allows all of the above, and adds physically simulated animation and audio (sound effect) behaviors. More importantly, we contribute the ability to add custom behavior types and new output modalities *without* requiring modifications to Elckerlyc's source code, described in Sections 4.3 and 4.4.

Integrating the realizer as a component in an application SmartBody offers integration with the Active MQ messaging system to provide independency of platforms and programming language, and to allow distributed setups. EMBR and Greta offer integration with the SEMAINE/Active MQ [8] messaging frameworks to achieve this; Greta additionally offers integration with Psyclone.

Elckerlyc uses Ports and Adapters to facilitate quick development of support for new types of integration; current implementations include support for the SEMAINE/Active MQ system and a simple direct TCP/IP connection. In Section 4.1 we discuss this in detail, and also touch upon several other things made possible by this architectural feature.

4 Design of a flexible and extensible BML Realizer

In this section we discuss the elements in Elckerlyc's architecture that facilitate configuration, extension, and adaptation of the system. We start with a global

overview. After that, we discuss the main possibilities in detail. For each topic we first sketch a 'user need'; subsequently, we show which elements of Elckerlyc are designed to meet that user need, and how one uses them.

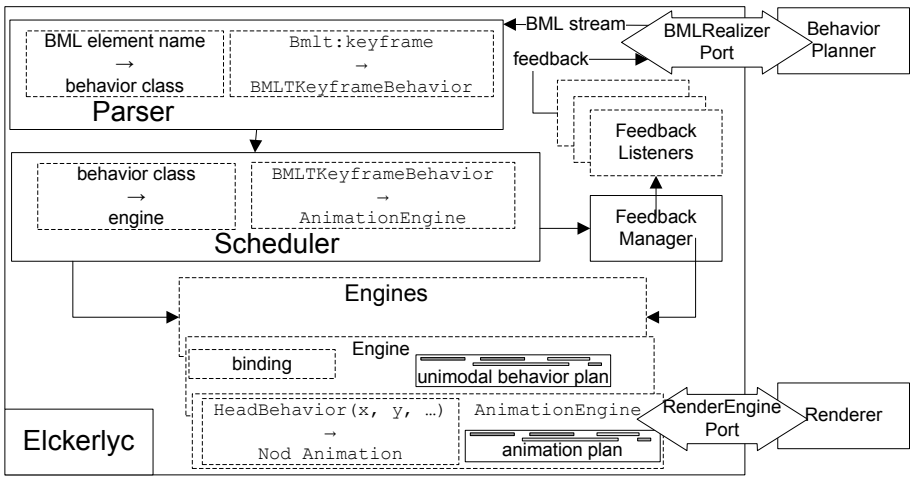

Fig. 1. Elckerlyc's architecture

Fig. 1 shows the relevant parts of the architecture. Dashed boxes indicate components that can be changed at initialization, black boxes indicate unchangeable components. The Behavior Planner controls the VH by sending a stream of BML Blocks to Elckerlyc through a BML Realizer Port. Section 4.1 discusses how Ports can be used, e.g., to integrate Elckerlyc with various distributed messaging systems. The Parser parses the BML stream, and provides the Scheduler with a list of BML behavior elements and time constraints between these elements. Section 4.3 discusses how to add custom BML behavior elements. The Scheduler generates an execution plan, based on these elements and constraints. Different Engines (e.g., a speech engine, an animation engine, a face engine) keep track of, and manage, unimodal plans for their specific modality. Section 4.4 discusses how to add new Engines. Engines are also responsible for translating behavior elements to a form that is actually displayed on the embodiment of the VH. Section 4.2 discusses how this mapping from abstract behavior element to concrete forms can be reconfigured. The final resulting animation is sent to the Renderer. Section 4.5 shows how new Renderers can be integrated with Elckerlyc.

4.1 Ports, Pipes, and Adapters

User need 1: Integrating Elckerlyc as component in an application

Elckerlyc is designed to be used as component in a larger application context. The application may need to run distributed over several machines, platforms,

and programming languages. The developer may want to log all interactions for post-hoc analysis. Nevertheless, the interface between Elckerlyc and application should remain as simple as possible: BML goes in; feedback comes out.

A minimal interface to a BML Realizer has functionality to (1) send a BML string to the Realizer and (2) register a listener for Realizer feedback. This is the BMLRealizerPort in Fig. 1. Both the Behavior Planner and the BML Realizer are connected to such a BMLRealizerPort. The adapter pattern [9] allows one to change the exact transport of BML and feedback to and from a BML Realizer, with no impact on the Behavior Planner and BML Realizer.

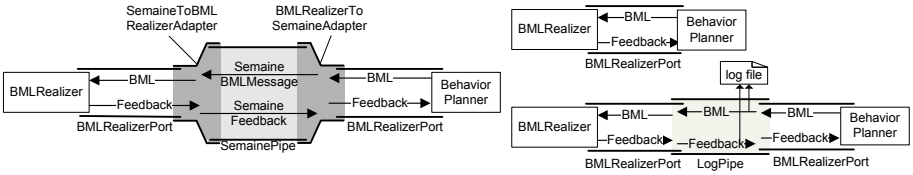

Fig. 2. Top right: the Realizer and BehaviorPlanner are connected directly on a RealizerPort. Left: the Realizer and BehaviorPlanner are connected through the Semaine API; they are unaware of this plumbing, they still communicate through RealizerPorts. Bottom right: a LogPipe logs the messages that pass through it to a file.

Elckerlyc implements the BMLRealizerPort interface. We have implemented Adapters that plug into BMLRealizerPorts and transport their messages over various messaging frameworks. Pipes are used to intercept BML and feedback, allowing one to measure it, let it go through slightly modified, or at a different rate. We have developed a pipe that logs the BML and feedback passing through, and one that buffers BML messages for a BMLRealizerPort that can only handle one BML message at a time. Fig. 2 shows some examples.

4.2 Gesture Binding and other Bindings

User need 2: Transparently Mapping BML to Output Behaviors

BML provides abstract behavior elements to steer the behavior of a VH. A specific BML Realizer is free to make its own choices concerning how these abstract behaviors will be displayed on the VH's embodiment. For example, in Elckerlyc, an abstract 'beat gesture' is by default mapped to a procedural animation from the Greta repertoire. The developer may want to map the same abstract behavior to a different form, e.g., to a high quality motion captured gesture.

Elckerlyc's AnimationEngine uses a XML description, called the GestureBinding, to achieve a mapping from abstract BML behaviors to Plan Units that determine how the behavior will be displayed in the embodiment. The GestureBinding, clearly illustrated in Fig. 3, can be customized by the application developer; other Engines provide similar bindings.

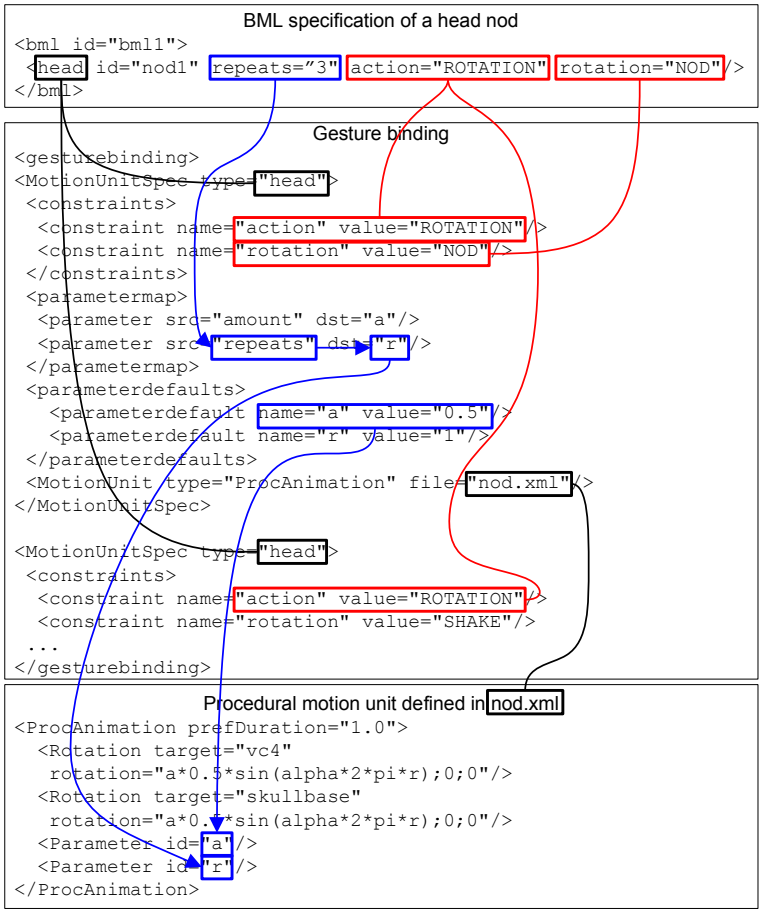

Fig. 3. Gesture Binding fragment binding the `head` element to the nod plan unit. Both the nod and shake motion units ex- ecute behaviors of type "head". They both satisfy the constraint action="ROTATION", but only the nod motion unit satisfies the constraint rotation="NOD" and is therefore selected to execute the head nod. The Gesture Binding maps the repeats parameter value in the BML behavior to the value of parameter `r` specified in the procedural motion unit. The value of parameter `a` is not defined in the BML head behavior, therefore the default value of `a`, as defined in the Gesture Binding, is used in the procedural animation.

4.3 BML Elements and Plan Units

User need 3: Adding new behavior types

Elckerlyc offers a large repertoire of Plan Unit types, in various Engines, that can be mapped in a Binding to give form to the abstract BML behaviors: physical simulation, procedural animation, morph target and MPEG-4 face control, Speech Units, etcetera. Still, a developer may need completely new Plan Unit types. For

example, to make the VH more lively, one may want to add a PerlinNoise Plan Unit that applies random noise to certain joints of the VH, as a kind of 'idle motion'. Such new Plan Units need to become available in the GestureBinding (see previous section); furthermore, one might want to extend the XML format of BML with < PerlinNoiseBehavior > to allow direct specification of this idle motion by the Behavior Planner.

New BML behaviors are created by subclassing the abstract class BMLBehaviorElement; they can be registered with the Parser using a static call. At initialization of Elckerlyc, the new BML behavior type are coupled to a single Engine by adding it to the behavior class → engine mapping (note that multiple behavior types can be coupled to the same Engine).

New PlanUnits implement the PlanUnit interface (for the AnimationEngine: rotate joints on the basis of time and animation parameters [2]). Such plan units are initialized from the GestureBinding through their class name (as a string), using Java's reflection mechanism (that is, the ability to construct a new object from its class name). This ensures that any Plan Unit implementing the right interface for an Engine can be used in the Binding for that Engine without requiring additional compile time dependencies.

4.4 New modality Engines

User need 4: Adding new modality Engines

The Nabaztag is a robot rabbit with ears that are controlled by servo motors and a body on which colored led lights are displayed. We needed to control this rabbit using BML, without encumbering Elckerlyc itself with Nabaztag specific code and libraries. To achieve this, we built a new Nabaztag Engine that was registered for handling all non-speech behaviors. For example, head nods were mapped in the Nabaztag Engine to a NabaztagPlanUnit that would move the ears shortly forward and back again; a sad face expression was mapped to a NabaztagPlanUnit that let the ears droop; etcetera.

Each Engine must implement the Engine interface (indicated by the lollipops in Fig. 4, top). All our current Engines are implemented on the basis of the DefaultEngine, a skeleton implementation of the interface. The DefaultEngine uses a Planner, PlanManager, Player and PlanPlayer and manages and plays a unimodal plan containing Plan Units (e.g. a gesture, a speech clause, etc.). The Planner resolves and constructs the unimodal plan on the basis of provided behavior elements and the constraints acting upon them. The PlanManager manages the unimodal plan and provides several functions to query its state or modify it. The Player plays back the units in the unimodal animation plan. In the DefaultPlayer, this functionality is fully delegated to a PlanPlayer. The Animation Engine and Face Engine require specialized Players that manage the combination of plan units that act simultaneously on the VH (e.g. physical simulation and keyframe animation), but can still delegate most of their playback functionality to a PlanPlayer. A MultiThreadedPlanPlayer plays its plan

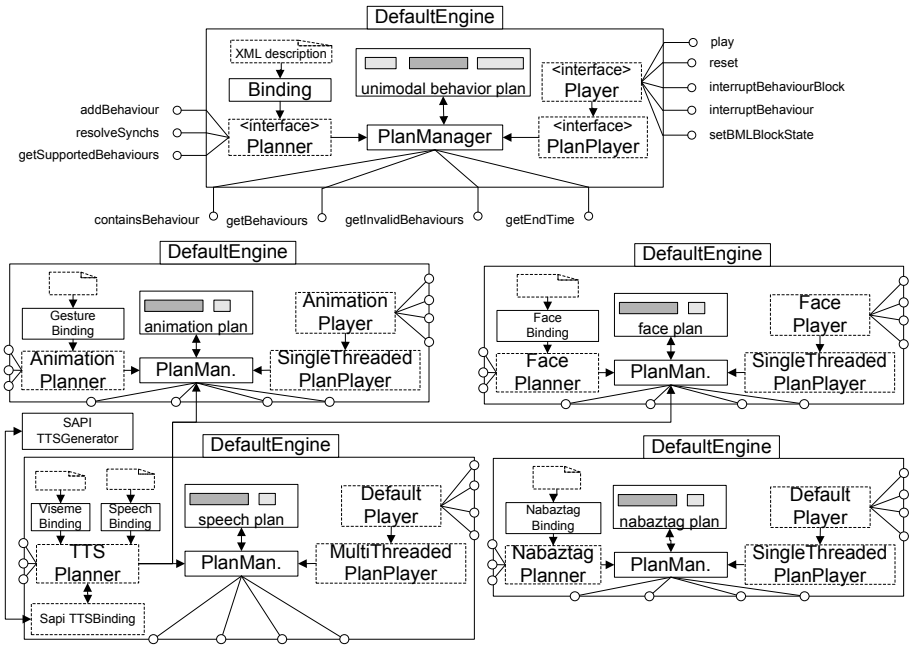

Fig. 4. Elckerlyc's Default Engine setup (top), and the internals of (from left to right, top to bottom) the Animation Engine, Face Engine, Speech Engine and Nabaztag Engine. Dashed blocks are changeable at initialization. Note that the Speech Engine requires access to the PlanManagers that handle the Animation and Face Plans, to set up the facial movement co-occurring with speech. The Nabaztag Engine, like most other Engines, mostly uses the default Engine components.

units in a seperate thread. This is beneficial for plan units whose playback would otherwise block the playing thread.

The Nabaztag Engine Building the new Nabaztag Engine involves developing the Plan Units that implement the basic control for the modality. A Plan Unit defines a way to control the robot – using one of its control primitives, see below – over the duration from the start time till the end of the Plan Unit. The control primitives for the Nabaztag robot are (1) move the ears of the robot to a specified position, (2) move the ears forward or backward by a specified amount, and (3) set one of the LEDs to a certain color. We implemented two Plan Unit types. The "MoveEarTo" Plan Unit moves the ears to a specified position by linear interpolation during the duration of the Plan Unit. The "WiggleEarTo" Plan Unit interpolates the ear from its current position to the specified target position and back to the starting point, during the duration of the Plan Unit, using a sinoid interpolation. Given these Plan Units, and a NabaztagBinding for mapping BML behaviors to Nabaztag PlanUnits, the Nabaztag Engine is constructed using the standard available Engine components (see Fig. 4). A completely new modality Engine has been added by implementing two basic

control Plan Units and an XML Binding. Due to the setup of Scheduler and Engines, synchronisation between the new Nabaztag Units and other modalities –e.g., speech– is automatically handled by Elckerlyc and requires no further implementation effort.

4.5 Integration with renderers

User need 5: Integration with other rendering environments

By default, Elckerlyc renders the VH in its own OpenGL based rendering environment. One might, however, want to use Elckerlyc to animate an embodiment in another rendering environment such as Half Life, Ogre, or Blender.

To seperate the renderer from Elckerlyc, we follow a design similar to that proposed by Russel and Blumberg [10]. The Animation Engine animates a local copy of the joint setup of the VH. The joint rotations set by Elckerlyc are copied to the renderer regularly (typically each frame).

The renderer therefore needs to support functionality to (1) provide Elckerlyc with the joint structure of the VH at its initialization, and (2) provide Elckerlyc with means to copy joint rotations to the virtual human in the renderer. Both requirements should be satisfied in a manner independent of renderer and transport (e.g. through TCP/IP, function call, shared memory). We use the remote procedural call framework Thrift [6] to achieve this. We have designed a language independent interface (using Thrift's interface definition language) that a renderer should implement to achieve connectivity with Elckerlyc. This interface is automatically compiled to an interface in the target language of the renderer. The transport mode is chosen at initialization time. We have made a proof-of-concept implementation for the Ogre rendering environment.

5 Discussion

We have discussed how Elckerlyc can be tailored to the needs of specific applications, without requiring invasive modifications to Elckerlyc itself. Elckerlyc's flexibility has allowed us to connect it to a behavior planner using either the SEMAINE framework or simple function calls, and to switch between such connections with a simple configuration option. The logging port allowed us to easily record all communication with Elckerlyc for user experiments, by simply changing the wiring between the behavior planner and Elckerlyc. The BMLRealizerPort also allowed us to exchange both the realizer and the behavior planner very easily. We have designed several behavior planners that implements behavior planning of a VH and one that replaces the VH behavior planning by a generic Wizard of Oz interface. The ability to easily replace the BML Realizer and behavior planner is also valuable for testing. We have designed a mockup BML Realizer that allows us to test behavior planners rapidly. This mockup BML Realizer does not actually execute the BML behavior, but does provide

the behavior planner with appropiate BML feedback. We have also designed a behavior planner that tests realizer implementations. This behavior planner executes test BML scripts on the realizer and inspects if the realizer provides the appropiate feedback. Since this test behavior planner communicates with the realizer through the generic BMLRealizerPort, it can not only test any configuration of Elckerlyc, but also potentially test Realizers designed by other research groups (by writing an adaptor from the BMLRealizerPort to their input and output channels). Elckerlyc's ability to add new modalities has allowed us to hook it up with the Nabaztag rabbit (see also Section 4.4) and to steer this rabbit with generic BML commands. The Nabaztag extension was achieved in a matter of days and did not require any changes in the Elckerlyc's source code.[2]

Elckerlyc's extensibility is mainly achieved by a very flexible initialization stage. In this initialization stage, a desired setup of the Elckerlyc Realizer is constructed by combining and configuring different components that are provided by Elckerlyc's code base or by custom extensions. We have designed an XML configuration file format that describes such a configuration. Several default configurations are available, and new configurations are typically easily achieved by slight modifications of an existing configuration.

References

1. 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: Intelligent Virtual Agents. Volume 4133 of LNCS., Springer (2006) 205–217
2. 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) (2010) 271–284
3. Thiebaux, M., Marshall, A.N., Marsella, S., Kallmann, M.: Smartbody: Behavior realization for embodied conversational agents. In: Autonomous Agents and Multiagent Systems. (2008) 151–158
4. Heloir, A., Kipp, M.: Real-time animation of interactive agents: Specification and realization. Applied Artificial Intelligence **24**(6) (2010) 510–529
5. Mancini, M., Niewiadomski, R., Bevacqua, E., Pelachaud, C.: Greta: a saiba compliant eca system. In: Troisiéme Workshop sur les Agents Conversationnels Animés. (2008)
6. Slee, M., Agarwal, A., Kwiatkowski, M.: Thrift: Scalable cross-language services implementation (2007)
7. Ekman, P., Friesen, W.: Facial Action Coding System: A Technique for the Measurement of Facial Movement. Consulting Psychologists Press, Palo Alto (1978)
8. Schröder, M.: The SEMAINE API: Towards a standards-based framework for building emotion-oriented systems. Advances in Human-Computer Interaction **2010**(319406) (2010)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Adisson-Wesley (1995)

[2] See `http://hmi.ewi.utwente.nl/showcase/elckerlyc` for screenshots and movies.

10. Russell, K.B., Blumberg, B.M.: Behavior-friendly graphics. In: Computer Graphics International, IEEE Computer Society (1999) 44–50