AsapRealizer in Practice — A Modular and Extensible Architecture for a BML Realizer

Dennis Reidsma[a], Herwin van Welbergen[b]

[a] *University of Twente, Department of Human Media Interaction, Netherlands*
[b] *University of Bielefeld, CITEC, Sociable Agents group, Germany*

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 one does this, one needs to be able to adapt and extend the capabilities of the virtual human as offered by the platform, without having to make invasive modifications to the platform itself. This is not trivial to support, and not all existing platforms facilitate this equally well. This paper describes how AsapRealizer (successor to Elckerlyc), a novel platform for controlling virtual humans, offer these possibilities.

Keywords: Elckerlyc, AsapRealizer, Behavior Markup Language, Virtual Human, Embodied Conversational Agents, Architecture, System Integration, Customization

1. Introduction

Virtual Humans (VHs) are commercially used in many educational and entertainment settings: serious gaming, interactive information kiosks, kinetic and social training, tour guides, storytelling entertainment, tutoring, entertaining games, motivational coaches, and many more. Researchers work with VHs to investigate the impact of specific social and communicative behaviors on the perception that users have of the VH, and the impact of a VH on the effectiveness and enjoyability with which tasks are completed. Building a complete VH from scratch is a daunting task, and it makes sense to

Email addresses: `d.reidsma@utwente.nl` (Dennis Reidsma), `hvanwelbergen@techfak.uni-bielefeld.de` (Herwin van Welbergen)

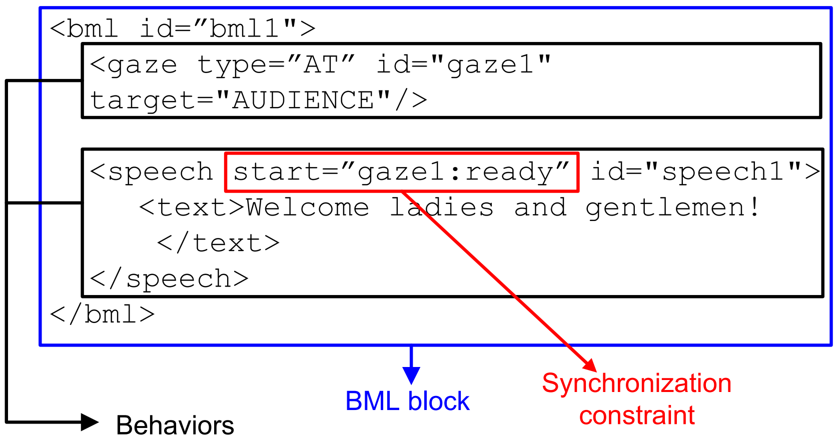
rely on existing platforms, for researchers and commercial developers both.

However, when one builds a novel interactive VH application, using existing platforms has its own drawbacks. One often needs to be able to adapt and extend the capabilities of the VH offered by the platform, and not all existing platforms facilitate this equally well. Specific additional gestures and face expressions might be needed; the application might need to run distributed over several machines; an 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 graphical embodiment of the VH might need to reside in a custom game engine; and one might need to plug in completely new custom behaviors and modalities for a specific usage context. Furthermore, all of these extensions and adaptations should be made without having to make invasive modifications to the platform itself. This last point is crucial, and will be worked out in more detail in the next chapter.

AsapRealizer, successor to Elckerlyc, is a state-of-the-art Behavior Realizer for virtual humans. Elsewhere, we described Elckerlyc's 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 on-the-fly 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 [1]. AsapRealizer [2] has been developed to combine these advantages of Elckerlyc with the incremental scheduling and co-articulation capabilities of ACE [3]. In thyis paper, we will focus on the role of AsapRealizer as a component in a larger application. We discuss how one can adapt AsapRealizer to suit the needs of a particular application, without giving up the level of abstraction offered by the BML Realization interface, and without having to modify the core AsapRealizer system itself.

2. Requirements for a Modular and Extensible Virtual Human Platform

A virtual human does generally not function in isolation: rather, they need to fulfill a role in a larger application context. The SAIBA framework [4] provides a good starting point for integrating interactive VHs in a larger system. Its Behavior Markup Language (BML, see Fig. 1) allows an application to specify the form and relative timing of the behavior (e.g. speech, facial



<bml id="bml1">
<gaze type="AT" id="gaze1" target="AUDIENCE"/>
<speech start="gaze1:ready" id="speech1">
<text>Welcome ladies and gentlemen!
</text>
</speech>
</bml>
BML block
Synchronization constraint
Behaviors

Figure 1: An example BML script with two BML behavior elements

expression, gesture) that a BML Realizer should display on the embodiment of a VH.

Although this level of abstraction, and the existence of several modern BML Realizers, saves a tremendous amount of effort when building new VH applications, BML does not provide the level of control over all details of the VH that is required by many applications. To those who need access to such details the BML Realizer should therefore not be a black box system, but allow access to such details. Yet, these details, configuration options and possibilities for extension should not add complications for people who do not need them.

2.1. Extensions and Modifications should be Non-Invasive

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

Ideally, a *non-invasive* extension or modification to a BML realizer only involves adding new *run-time libraries* or new *resources* to the classpath, and should not require *compile time* dependencies for the BML realizer on new code. This requirement is the driving force behind many of the architectural choices described later in this paper.

2.2. Requirements for Extensibility and Configurability

Below follows a number of extensibility requirements for BML realizers that should be implemented as non-invasive modifications. In the next chapter we will explain each requirement in more detail and show how each was solved in Elckerlyc and AsapRealizer; after that, we will compare our solutions to related state-of-the-art systems.

- It should be possible to integrate new renderers
- It should be possible to integrate new speech synthesizers
- It should be possible to integrate new physics simulators
- Transport of the BML stream to the realizer should be flexible and configurable
- It should be possible to adapt the BML stream with capabilities for filtering and logging
- The realizer needs a transparent and configurable mapping from input (BML behavior elements) to output (control of the VHs embodiment)
- It should be easy to add new behavior types or output modalities
- The realizer needs the capability to be integrated as a component in application, independent of variables such as the OS and programming language on which the application is developed
- It should be easy to run the realizer as part of a setup distributed over multiple machines
- The realizer requires the possibility (and tools) to add new assets such as new 3D models or new behavior repertoire (e.g., animations and face expressions)

The BML realizer Elckerlyc facilitates all these possibilities for modification and extension without requiring invasive modification to Elckerlyc itself. Its successor AsapRealizer additionally meets the following requirements:

- It should be possible to change the BML Scheduler, as BML Scheduling is non-trivial, especially when one wants to allow on-the-fly modification of plans [5]; we need flexibility to experiment with new scheduling algorithms and compare them with the existing ones

- It should be possible to hook up different lipsync modules to the same TTS system, allowing lipsynch on different embodiments (e.g., robot, avatar or jpeg picture) and allowing one to experiment with different lipsync algorithms (e.g., various co-articulation solutions, visual emotional speech)

- It should be possible to configure the whole setup of a virtual human in run-time, preferably through easy-to-use configuration files (determining, e.g., which embodiments to use, which gesture repertoire to load, which lipsynch solution to apply, and many other things)

3. AsapRealizer: Basic Architecture

Before going into detail concerning the possibilities for extension and configuration, this chapter introduces the basic architectural concepts behind AsapRealizer. Fig. 2 shows a simplified view of its SAIBA architecture. The Behavior Planner controls the VH by sending a stream of BML Blocks (cf. Fig. 1) to AsapRealizer through a BML Realizer Port.[1] The BML Blocks describe what behavior the VH should display. AsapRealizer, as the BML Realizer, controls the embodiment of the virtual human to make this happen, and sends back feedback about the progress.

Fig. 3 shows the parsing and routing of BML blocks in more detail. The Parser parses the BML stream, and provides the Scheduler with a list of behaviors and time constraints between these behaviors.[2] 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. The Scheduler uses a configurable mapping to determine which Engine must handle

[1]Section 4.1 discusses how Ports can be used, e.g., to integrate AsapRealizer with various distributed messaging systems.

[2]Section 4.4 discusses how to add custom BML behavior elements, and how to register them with the Parser.

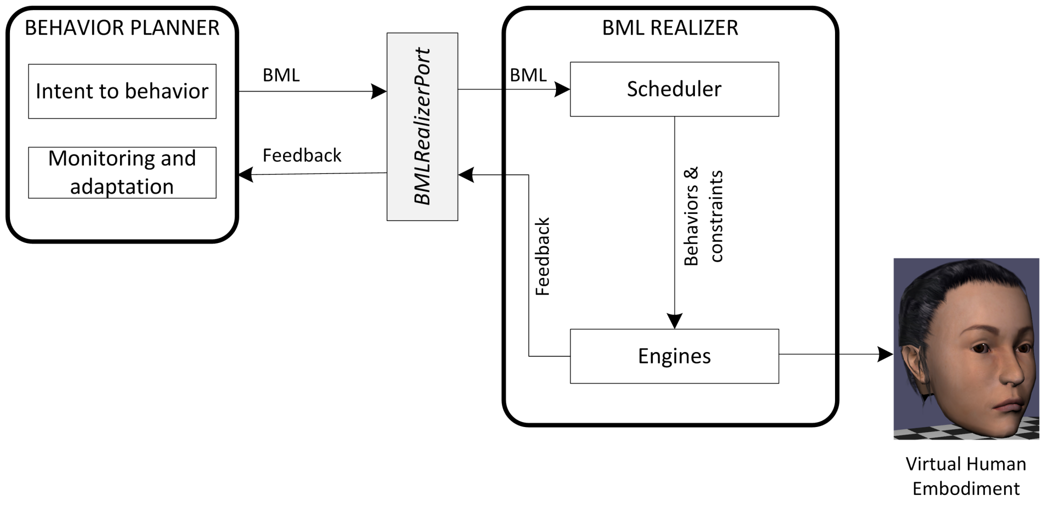


Figure 2: Global overview: Simplified SAIBA architecture[4]. A Behavior Planner constructs BML scripts that describe what behavior the VH should display; the BML Realizer controls the embodiment to make this happen; progress feedback is sent back to the Behavior Planner.

which of the behaviors.[3]

Fig. 4 shows that Engines are responsible for translating the behaviors and constraints to a form that is actually displayed on the Embodiment of the VH. Ultimately, behaviors are displayed on an Embodiment by accessing the *control primitives* of that Embodiment. A FaceEmbodiment is controlled by setting MPEG4 values; a SkeletonEmbodiment is controlled by rotating joints; etcetera. There may be multiple implementations of a specific Embodiment interface, offering exactly the same control primitives. For example, Fig. 5 shows two implementations of the FaceEmbodiment interface: a 3D graphical face where MPEG4 values (the control primitives of FaceEmbodiments) result in mesh deformations, and a 2D cartoon face where MPEG4 values lead to modification of the Bezier curves defining the elements of the face.

The control primitives of the Embodiments are accessed by Engine specific Plan Units; an Engine will indirectly control the embodiments by translating the behaviors and constraints into an unimodal plan of these Plan

[3]Section 4.5 discusses how to add new Engines.

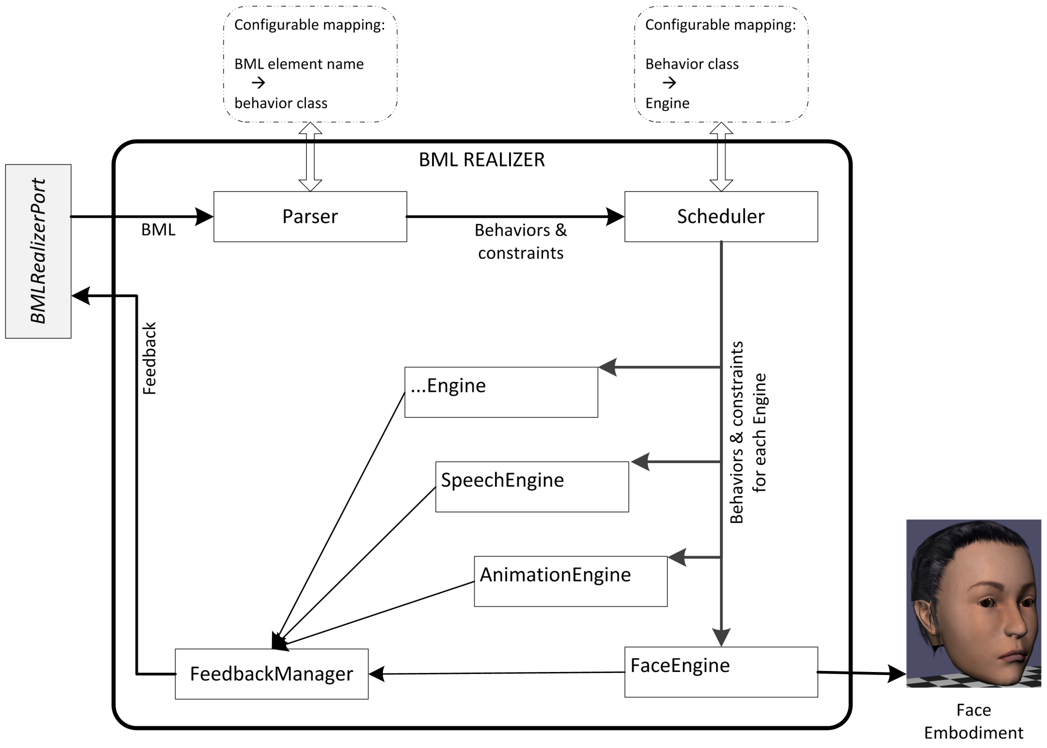


Figure 3: The parsing and routing of BML in AsapRealizer in more detail: Incoming BML blocks are parsed; the scheduler maintains the multimodal plan, distributing the behaviors and constraints over the various unimodal engines.

Units.[4] A FaceEngine maintains a plan of FacePlanUnits; each FacePlanUnit will modify the MPEG4 values of a FaceEmbodiment while it is being played. An AnimationEngine, in comparison, maintains a plan of AnimationMotionUnits; these will control a SkeletonEmbodiment by modifying its joint rotations.[5] An overview of various Engines, their Plan Units, and the types of Embodiment they control, can be found in Appendix A.

[4]Section 4.2 discusses how this mapping from abstract behavior element to concrete forms can be reconfigured.

[5]Section 4.3 discusses various types of embodiments available in AsapRealizer, and how to add new ones. Section 4.6 shows how graphical embodiments can reside in any render engine.

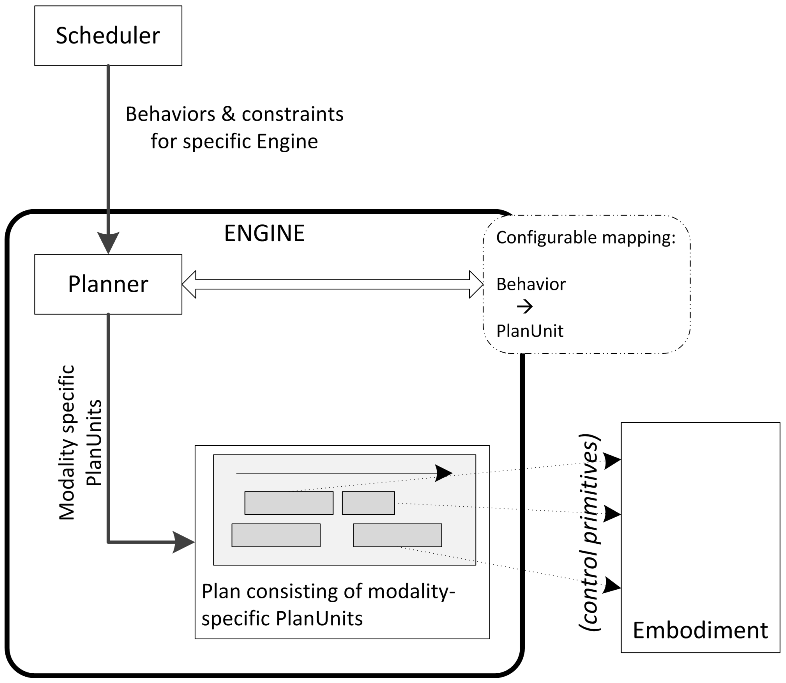

Scheduler
Behaviors & constraints for specific Engine
ENGINE
Planner
Configurable mapping:
Behavior → PlanUnit
Modality specific PlanUnits
Plan consisting of modality-specific PlanUnits
(control primitives)
Embodiment

Figure 4: Realizing the behaviors on an Embodiment: A unimodal Engine is responsible for translating the behaviors into a form that can be displayed on the embodiment of the virtual human.

4. Solutions for a Flexible and Extensible BML Realizer

The figures in the previous chapter already indicate a few points where AsapRealizer allows for easy configuration and extension. In this chapter we discuss in more detail the elements in AsapRealizer's architecture that facilitate configuration, extension, and adaptation of the system. For each topic we first sketch a 'user need'; subsequently, we show which elements of AsapRealizer are designed to meet that user need.

4.1. Ports, Pipes, and Adapters

User need 1: Connecting the application to AsapRealizer

AsapRealizer 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 AsapRealizer and application should remain as simple as possible: BML goes

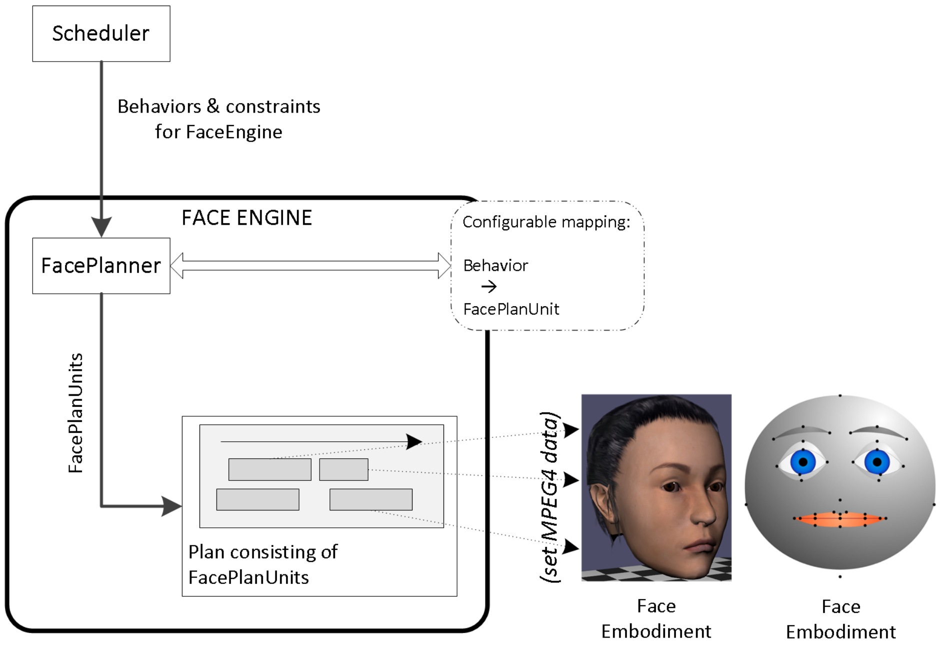


Figure 5: A FaceEngine can control any kind of FaceEmbodiments by accessing its control primitives (i.e., setting MPEG4 values).

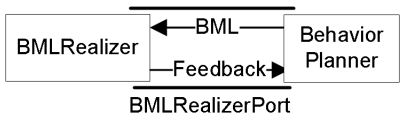


Figure 6: The BML Realizer and Behavior Planner are connected directly on a Realizer-Port.

in; feedback comes out. Adding logging, network transport, and such, should not be noticable in how AsapRealizer and the application communicate with each other.

A minimal interface between application and 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. 2 —displayed in isolation in Fig. 6. The Behavior Planner and the BML Realizer are connected to the front and back end of such a BMLRealizerPort. The adapter pattern [6] 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.

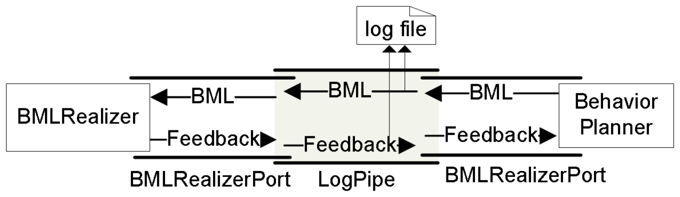


Figure 7: A LogPipe logs the messages that pass through it to a file.

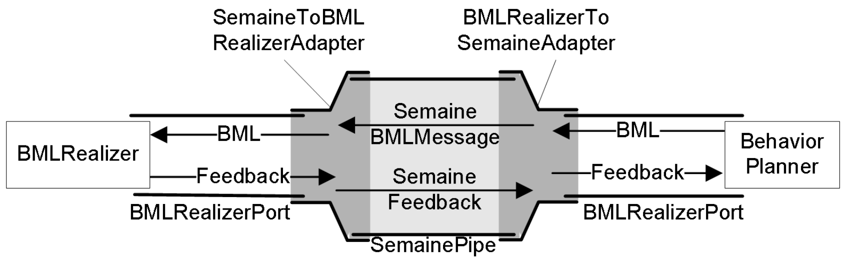


Figure 8: The Realizer and BehaviorPlanner are connected through the Semaine API; they are unaware of this plumbing, they still communicate through RealizerPorts. Similar implementations already exist for ActiveMQ and for direct TCP/IP transport.

AsapRealizer implements the BMLRealizerPort interface. We have implemented Adapters that plug into BMLRealizerPorts and transport their messages over various messaging frameworks (Fig. 8). Pipes are used to intercept the BML messages and the 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 (Fig. 7), and one that buffers BML messages for a BMLRealizerPort that can only handle one BML message at a time.

4.2. Gesture Binding and Other Bindings

User need 2: Transparently Mapping requested Behaviors to PlanUnits

BML provides BML 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 AsapRealizer, an abstract 'beat gesture' is by default mapped to a procedural animation from the repertoire of the Greta realizer by Pelachaud and her team (see also Section 5). The developer may want to map the same abstract behavior to a different form, e.g., to a motion captured gesture.

In AsapRealizer, unimodal Engines are responsible for mapping the requested Behaviors to PlanUnits (cf. Fig. figPlanUnitAndEmbodimentGeneric). In AsapRealizer, XML files called *Bindings* are used to allow one to configure this mapping. AsapRealizer's AnimationEngine uses the GestureBinding to achieve a mapping from the behaviors and constraints (delivered by the Scheduler) to Animation Plan Units that determine how the behavior will be displayed on the embodiment. The GestureBinding XML file, clearly illustrated in Fig. 9, can be customized by the application developer; other Engines provide similar bindings.

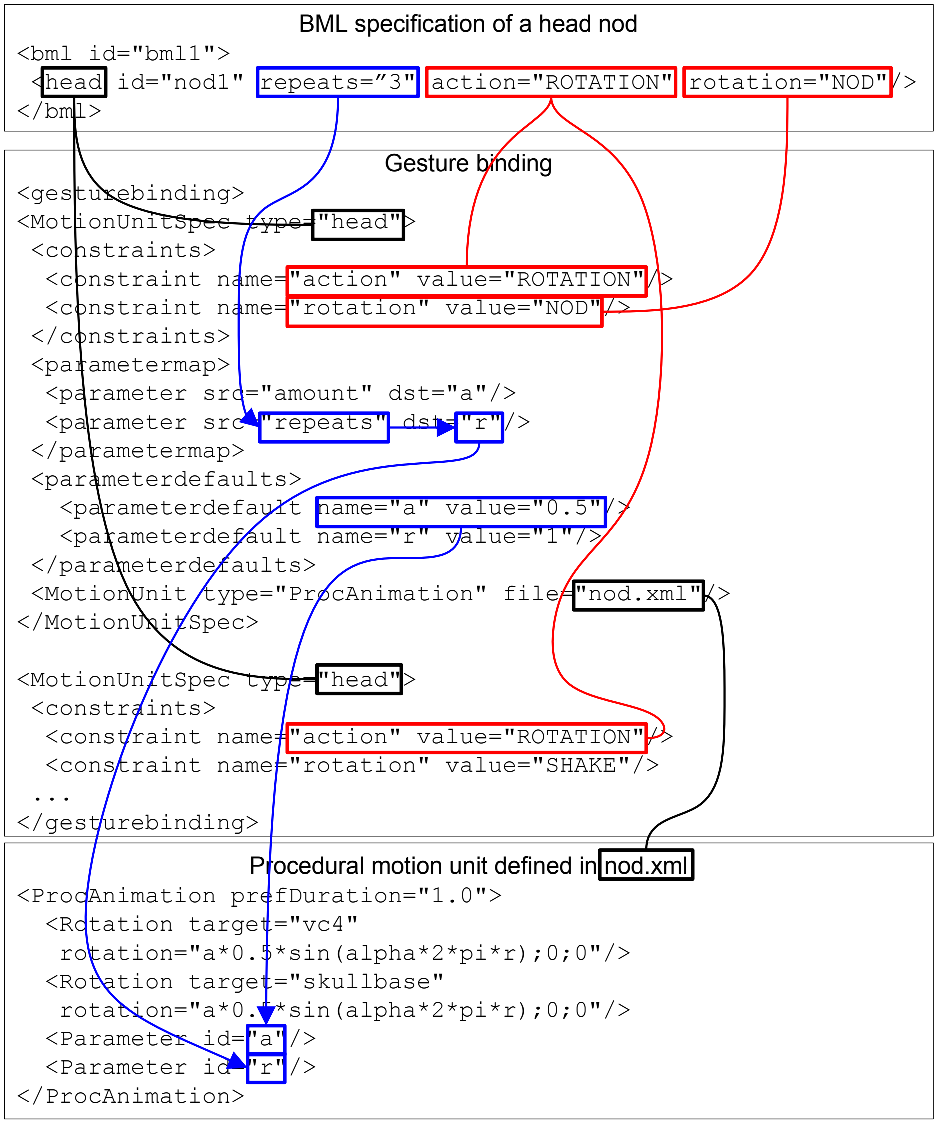
4.3. New Embodiments

User need 3: Adding new Embodiments

A specific Engine should be able to control any new type of VH body that offers the right type of control, without necessitating changes to the implementation of the Engine. For example, the default AnimationEngine could conceivably control any VH body that allows for joint rotations in an H-Anim hierarchy. For this to succeed, these bodies all need to offer the same interface with the same control primitives to the AnimationEngine.

Each type of Engine in AsapRealizer is designed to control an Embodiment that implements a specific interface. An AnimationEngine requires its Embodiments to implement the SkeletonEmbodiment interface. The control primitives for that type of Embodiment allow setting joint rotations in an H-Anim joint hierarchy. Current implementations include one controlling an avatar in our own OpenGL rendering envirorpment, one controlling (through Thrift [7]) an avatar in Ogre, and one controlling an avatar residing in a Re-Lion SUIT environment. A FaceEngine requires its Embodiments to implement the FaceEmbodiment interface, which offers control primitives for Mpeg-4 facial animation parameters. Current implementations of this interface include one controlling avatars in our own OpenGL rendering environment, a Mpeg-4 controlled 2D cartoon face, and an implementation that allows our FaceEngine to control an XFace talking head [8]. Other Engines have their own required Embodiments.

For each new implementation of a certain Embodiment interface to be used in our Engines, a loader class needs to be implemented that makes the Embodiment type available in our XML based virtual human loader system. This loader system will be discussed in a later secction.


BML specification of a head nod
Gesture binding
Procedural motion unit defined in nod.xml

```
<bml id="bml1">
 <head id="nod1" repeats="3" action="ROTATION" rotation="NOD"/>
</bml>
```

```
<gesturebinding>
<MotionUnitSpec type="head">
 <constraints>
  <constraint name="action" value="ROTATION"/>
  <constraint name="rotation" value="NOD"/>
 </constraints>
 <parametermap>
  <parameter src="amount" dst="a"/>
  <parameter src="repeats" dst="r"/>
 </parametermap>
 <parameterdefaults>
   <parameterdefault name="a" value="0.5"/>
   <parameterdefault name="r" value="1"/>
 </parameterdefaults>
 <MotionUnit type="ProcAnimation" file="nod.xml"/>
</MotionUnitSpec>

<MotionUnitSpec type="head">
 <constraints>
  <constraint name="action" value="ROTATION"/>
  <constraint name="rotation" value="SHAKE"/>
 ...
</gesturebinding>
```

```
<ProcAnimation prefDuration="1.0">
  <Rotation target="vc4"
   rotation="a*0.5*sin(alpha*2*pi*r);0;0"/>
  <Rotation target="skullbase"
   rotation="a*0.5*sin(alpha*2*pi*r);0;0"/>
  <Parameter id="a"/>
  <Parameter id="r"/>
</ProcAnimation>
```

Figure 9: Gesture Binding fragment binding the `head` element to the nod Plan Unit. Both the nod and shake motion units execute 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.

4.4. BML Elements and Plan Units

User need 4: Adding new behavior types

The various Engines in AsapRealizer offer a large repertoire of Plan Unit types 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 existing or newly developed types of Embodiments. For example, to make the VH more lively, one may want to add a Perlin-Noise 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 elements are created by subclassing the abstract class BML-BehaviorElement; they can be registered with the Parser using a static call: `BMLInfo.addBehaviourType(xmltag, BehaviorElementClassName);`

At initialization of AsapRealizer, the new BML behavior type are coupled to a single Engine by adding it to the `behavior class` $\rightarrow$ `engine` mapping (see Fig. 3; note that multiple behavior types can be coupled to the same Engine). This can also be done through a static call, or by adding a `<Routing>` section to the VH loader XML file (see Fig. 15).

New Plan Units implement the appropriate subinterface of the PlanUnit interface (for the AnimationEngine: MotionUnits that rotate joints on the basis of time and animation parameters [1]). 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.

4.5. New Modality Engines

User need 5: 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 AsapRealizer 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, BML head nods were mapped in the NabaztagBinding 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.*

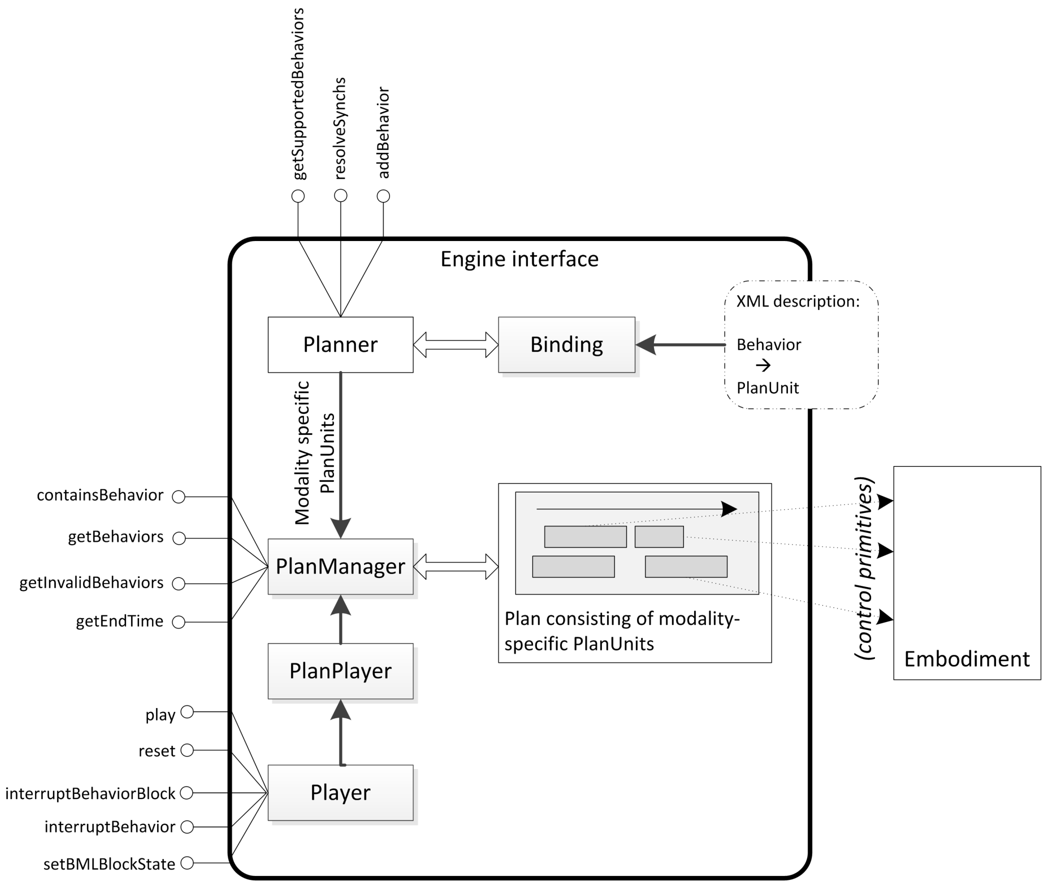


Figure 10: AsapRealizer's Engine Interface. Dashed blocks are changeable at initialization.

To facilitate development of new Engines, a series of default implementations of many of the necessary components are available. One generally has to re-implement only very few of these components to achieve a complete new Engine. As already discussed in Chapter 3, an Engine needs to maintain a modality-specific plan. Fig. 10 shows this in more detail. The data structure for the unimodal plan is maintained by a default PlanManager that provides several functions to query its state and modify it. Playing the plan (i.e., executing the PlanUnits on the embodiment) is coordinated by a Player, that generally delegates this to one of the default PlanPlayer implementations (a multi threaded PlanPlayer when calls to the Embodiment's control primitives are blocking, and a single threaded Planplayer, otherwise). Implementing a new Engine usually requires three steps: implementing PlanUnits specifically for this Engine (cf. Fig. 4), implementing a Binding to easily define a configurable mapping from behaviors to these PlanUnits, and implementing a Planner specialized in constructing plans for this modality. The Animation Engine and Face Engine in addition 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 DefaultEngine implementation of the Engine interface encapsulates these elements, connects them to each other, and provides the BMLRealizer with access to their functionality to the BML Realizer through the Engine interface.

4.5.1. 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. 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 AsapRealizer and requires no further implementation effort.

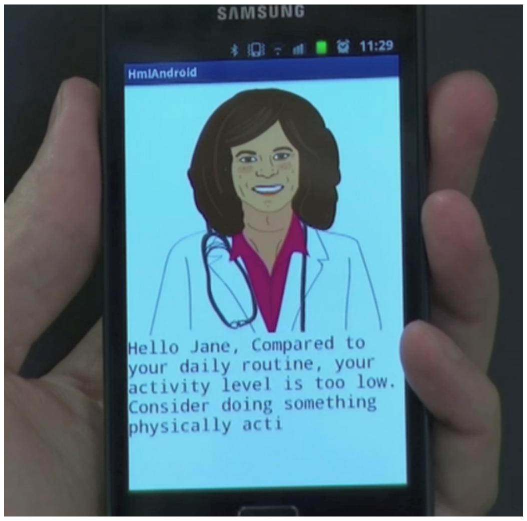

SAMSUNG
11:29
HmlAndroid
Hello Jane, Compared to your daily routine, your activity level is too low. Consider doing something physically acti

Figure 11: The PictureEngine running on an Android smartphone.

4.5.2. Other Engines

We have implemented a variety of other useful engines in AsapRealizer. The TextEngine, for example, can be used to re-route speech behaviors, so they are not realized by text-to-speech synthesis. Instead, the TextEngine employs TextUnits that display the text string representing the speech on a TextEmbodiment (e.g., console output, a text label in the GUI, cartoon text balloons, etcetera). The PictureEngine is particularily useful for building cartoon agents: it allows one to have BML behaviors realized as a series of layered pictures, instead of a skeleton animation. The PictureEngine has been employed in the Smarcos project to port the AsapRealizer BML Realizer to an Android Smartphone that did not have enough processing power for displaying full 3D OpenGL based graphics (see Fig. 11) [9]. The NaoEngine is implemented in a way similar to the NabaztagEngine, to control a Nao Robot[6] using AsapRealizer. We are currently working on implementing more Engines for various robotic embodiments.

4.6. Integration with Renderers

User need 6: Integration with other rendering environments

[6]`http://www.aldebaran-robotics.com`

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

Integration of AsapRealizer with any new renderer is simply a matter of adding an implementation of the SkeletonEmbodiment interface that communicates joint rotations (as set by AsapRealizer) to the graphical avatar displayed in the renderer, and one for the FaceEmbodiment (communicating MPEG4 values). The SkeletonEmbodiment needs to support functionality to (1) provide AsapRealizer with the joint structure of the VH at its initialization, and (2) provide AsapRealizer with means to copy joint rotations to the virtual human in the renderer. The FaceEmbodiment needs only to provide AsapRealizer with means to set MPEG4 values. These 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 [7] 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 AsapRealizer. 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, and for Re-Lions[7] SUIT environment.

4.7. Text-To-Speech Generation and Speech Scripts

User need 7: Integration with text-to-speech synthesis systems

---

Different applications might have different requirements for the text-to-speech voices. Another language, another TTS system, another markup language offering control of exactly the right voice features at times all need to be integrated with a VH configuration.

Speech for virtual humans can be generated using various Text-To-Speech systems. Furthermore, each TTS system may be able to use several standardized (e.g. SSML) or vendor-specific (e.g. MaryTTS, Microsoft Speech

[7]`http://www.re-lion.com`

API, Fluency TTS) speech description languages that allow one to change features of the produced speech.

AsapRealizer provides the extensibility to easily hook up new TTS generators, and new TTS specification languages having their own XML markup format. To use a certain speech engine, and to allow it to generate speech using a specific speech description language, one should implement a TTSBridge for that language and engine. This TTSBridge provides a standardized interface to 1) speak a string, 2) store speech specified in a string to a file, and 3) get timing information on a to-be-spoken string. These strings should contain the speech text, specified in the specific language for that bridge. The speech engine is set up at initialization time with a TTSBinding that maps a specific extension of a speech behavior ("markup language") to a specific TTSBridge. TTSBridges are currently implemented for default BML speech in Mary TTS, default BML speech in Microsoft Speech API, SSML in Mary TTS, SSML in Microsoft Speech API, SAPI XML in Microsoft Speech API, several Mary TTS XML formats, and default BML speech using the Android speech synthesis SDK. Default TTSBindings are also available for MaryTTS and Microsoft Speech API. These default TTSBindings map all speech behaviors the TTS generator supports to matching TTSBridges for that TTS generator. Adding more TTSBridges and TTSBindings is a matter of implementing the right interfaces and adding the resulting jar file to the classpath – AsapRealizer will automatically pick up and make available the new voices and speech markup languages.

4.8. Lipsync

User need 8: Setting up lipsync

The same TTS-system may have to be used to steer the lips of very different embodiments (robots, virtual humans, 2D cartoon figures), using similar control primitives (e.g. visemes). Additionally, users might want to experiment with different lipsync algorithms for the same embodiment, or apply a specialized lipsync strategy on a custom face.

Lip motion can be generated using a wide array of specific motion engines (e.g. the AnimationEngine, FaceEngine or specialized engines for motion on robots or 2D characters). Rather than directly coupling the SpeechEngine to such motion engines, the SpeechEngine steers one or more LipsyncProviders that are registered to it. Implementations of a LipsyncProvider (see Fig. 12 for its interface) then steer lip motion on their specific engine, with the

```
public interface LipsynchProvider
{
    void addLipSyncMovement(BMLBlockPeg bbPeg, SpeechBehaviour beh,
        TimedPlanUnit speechUnit, List<Visime> visemes);
}

public final class Visime
{
    private final int duration;   //duration in ms
    private final int number;     //visime number
    private final boolean stress; //stressed in current word?
    ...
}
```

Figure 12: The interface of a LipSyncProvider and the Viseme value class. Lipsync motion is requested per speechbehaviour, given the bml block, the TimedPlanUnit of the accompanying speech and a list of Visemes.

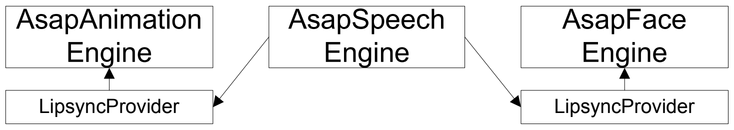
desired lip sync algorithm. The registration of selected LipsyncProviders on the SpeechEngine is arranged at initialization time. In Fig. 13 we illustrate a typical configuration for a 3D virtual human.

4.9. Loading a Complete System

User need 9: Connecting everything together

A system that offers so many ways of extending and modifying the capabilities of the virtual human may be very confusing to initialize. How does one connect all the modules together? Load the correct embodiments? Connect them to the appropriate Engines?

AsapRealizer offers many ways to extend the system. Add one library to the classpath, and you have a new type of Embodiment available that, for example, allows you to control an avatar in the Half Life rendering environment. Add another library to the classpath, and you suddenly have a new TTS system available for the SpeechEngine, are able to control your new robot using BML, etcetera. In addition, there are several so-called “Environments” available: a default render environment in which you can load and render avatars that are controlled by the various Engines, a physics environ-



```
<Loader id="animationengine" ... </Loader>
<Loader id="faceengine" ... </Loader>
<Loader id="facelipsync" requiredloaders="faceengine"
    loader="asap.faceengine.loader.
    TimedFaceUnitLipSynchProviderLoader">
    <MorphVisemeBinding resources="Humanoids/armandia/facebinding/"
    filename="ikpvisemebinding.xml"/>
</Loader>
<Loader id="jawlipsync" requiredloaders="animationengine"
   loader="asap.animationengine.loader.
     TimedAnimationUnitLipSynchProviderLoader">
  <SpeechBinding basedir=""
  resources="Humanoids/shared/speechbinding/"
  filename="ikpspeechbinding.xml"/>
</Loader>
<Loader id="speechengine" loader="asap.speechengine.loader.
   SpeechEngineLoader"
   requiredloaders="facelipsync,jawlipsync">
    ...
</Loader>
```

Figure 13: Top: a common lipsync configuration for 3D virtual humans. The SpeechEngine is hooked up to a LipsyncProvider that connects to the AnimationEngine providing it with jaw joint rotations for lipsync, and with a LipsyncProvider that connects to the FaceEngine and provides facial animation through blend shapes. Bottom: the initialization as specified in a loader to achieve this particular lipsync configuration.

ment that does the physics simulation, an audio environment that takes care of playing voices and audio for multiple VHs, etcetera.

In order to facilitate easy initialisation and configuration of the various Engines and Embodiments, AsapRealizer offers an Environment package with support for developing Engines and Embodiments and for loading them and connecting them with each other, and a generic XML based loader package with support for configuring a complete VH setup using one XML file. The latter depends on the Java reflection mechanism for identifying the loader classes present in the libraries for every type of Embodiment and Engine. Fig. 14 shows an example code fragment setting up a VH in an Environment and Fig. 15 an XML fragment of a VH loader specification.

```
//create all required environments
RenderEnvironment hre = new RenderEnvironment();
OdePhysicsEnvironment ope = new OdePhysicsEnvironment();
MixedAnimationEnvironment mae = new MixedAnimationEnvironment();
AudioEnvironment aue = new AudioEnvironment();
AsapEnvironment ae = new AsapEnvironment();

ArrayList<Environment> environments = new ArrayList<Environment>();
environments.add(hre);
environments.add(ope);
environments.add(mae);
environments.add(aue);
environments.add(ae);

//initialize the environments
hre.init();
ope.init();
aue.init();
mae.init(ope);
ae.init(environments);

//after initialisation: start the physics and animation clocks
hre.startRenderClock();
ope.startPhysicsClock();

//load a virtual human
```

Figure 14: Example code fragment loading and starting the Environment in which a Virtual Human will be loaded.

```
<Loader id="realizer"
        loader="realizerembodiments.AsapRealizerEmbodiment">
  <PipeLoader id="logpipe" loader="bmlpipe.LogPipeLoader">
    <Log requestlog="saiba.bml.requests"/>
  </PipeLoader>
  <PipeLoader id="activemqpipe" loader="bmlpipe.ActiveMQPipeLoader"/>
  <ServerAdapter requestport="7521" feedbackport="1257"/>
</Loader>

<!-- graphical embodiment implements both SkeletonEmbodiment
     and FaceEmbodiment -->
<Loader id="graphicalembodiment"
        loader="renderenvironment.HmiRenderEmbodimentLoader">
  <Body filename="collada_avatar.bin"/>
</Loader>

...

<Loader id="animationengine"
        loader="animationengine.MixedAnimationEngineLoader"
        requiredloaders="mixedskeletonemb,physicalemb">
  <GestureBinding filename="gesturebinding.xml"/>
</Loader>

<Loader id="faceengine"
        loader="faceengine.FaceEngineLoader"
        requiredloaders="graphicalembodiment">
  <FaceBinding filename="facebinding.xml"/>
</Loader>

<Loader id="speechengine"
        loader="speechengine.SpeechEngineLoader"
        requiredloaders="faceengine,animationengine">
  <Voice voicetype="SAPI5" voicename="Kate"/>
</Loader>

<!-- This overrides the default Engine routing -->
<BMLRouting>
  <Route behaviourclass="saiba.bml.core.FaceBehaviour"
         engineid="faceengine"/>
</BMLRouting>
```

Figure 15: Partial XML specification for loading a VH setup

4.10. Conflict Resolution

User need 10: Conflict resolution

Multiple engines (or the same engine) might steer the same embodiment simultaneously in different manners. How are conflicts between such engines managed?

The complete freedom that the above XML Loader mechanism offers, leads to the distinct possibility that there are multiple Engines all claiming to steer the same Embodiment. For example, the FaceEmbodiment might be steered both by the FaceEngine for face expressions, and by a WizardOfOzEngine in which the eye rotations and blinking are controlled based on the eye movements of a human operator. In AsapRealizer, such conflicts are currently handled on one of two ways. Firstly, the class that implements the Embodiment interface might itself have mechanisms to recognize conflicting demands, and resolving them by giving higher priority to certain types of requests, or actively blending the result of two conflicting requests. The standard FaceEmbodiment implementation, for example, will add MEG4 requests from multiple sources to each other, resulting in a cumulative effect on the MPEG4 control points. Secondly, an Embodiment interface might offer a way in which an Engine can exclusively claim a certain control primitive, thereby actively disallowing other Engines to use that particular control primitive. The latter mechanism is used in AsapRealizer to prevent the AnimationEngine and the WizardofOzEngine simultaneously attempting to rotate the neck of the virtual human.

4.11. Conclusion

In this chapter, we described in detail many of the architectural elements of AsapRealizer that facilitate non-invasive configuration and extension of the capabilities of a Virtual Human, that do not require recompilation of the core of AsapRealizer itself. Some elements involve only modification of resource files. The other changes only involve implementing a few interfaces and adding the resulting jar to the classpath – AsapRealizer will automatically pick up the new Embodiments, Engines, and capabilites as soon as they are referred to in the XML Loader file with which a new VH is loaded. In the next chapter, we will compare these aspects with the most prominent related work.

5. Comparison with other platforms

Table 1: Capabilities that can be changed without recompilation, per realizer.

	SmartBody	EMBR	Greta	ACE	Elckerlyc	AsapRealizer
Renderer	✔	✔	✔	✔	✔	✔
TTS system	✔	✘	✘	✔	✔	✔
BML transport wiring	✘	✘	✘	✘	✔	✔
BML to scripted output mapping	✔	✔	✔	✔	✔	✔
BML to procedural output mapping	✘	✘	✘	✘	✔	✔
Output modality	✘	✘	✘	✘	✔	✔
Behavior scheduling algorithm	✘	✘	✘	✘	✘	✔
Lipsync algorithm	✘	✘	✘	✘	✘	✔
Lipsync modality	✘	✘	✘	✘	✘	✔

Like AsapRealizer, the BML Realizers Smartbody [10], EMBR [11] and Greta [12] were specifically designed for integration with new and existing renderers, to allow a wide range of behavior types, to provide tools for asset creation, and/or to facilitate integration in a larger application setup. AsapRealizer additionally contributes a transparent and adjustable mapping from BML to procedural output (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 addition to this, AsapRealizer provides the unique capability to plug in new lipsync algorithms, lipsync modalities, and scheduling algorithms, without recompilation of its core.

In this section, we discuss if and how various requirements were solved for the three realizers mentioned above, and shortly indicate the differences with our solutions. Clearly, there are more reasons to prefer one realizer over another than just configurability and ease of integration. For example, a aprticular strong point of SmartBody is their high quality animations; Greta is well known for its state-of-the-art face expression control and especially its emotional visual speech generation; AsapRealizer is particularily suitable for applications that require anytime, on-the-fly, adaptation of generated behavior; and EMBR offers a detailed specification language for procedural animation, facilitating precise reproduction of annotated gestures. In this paper, though, we focus mostly on the requirements related to configurability, adaptability, and ease of integration into larger application contexts, especially for users who want to use a BML realizer, but are not developers of realizers themselves. Table 1 provides an overview of the comparison detailed in the rest of this section.[8]

5.1. Integration with New and Existing Renderers

When a VH is used in an application, it may need to reside in a 3D world running in any (new or existing) render engine. A BML Realizer should therefore be able to control avatars in such a render engine as easily as its 'standard embodiments'.

Smartbody provides the BoneBus library to connect the Smartbody realizer to a renderer. BoneBus uses UDP to transport (facial and skeletal) 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. SmartBody has been integrated with the Unreal 2.5 and Panda3D (in CADIA's branch of Smartbody) renderers; partial integrations are available for Gamebryo, Half-Life 2 and Ogre.[9]

In EMBR, the renderer is seperated from the realizer, and both provide a representation of the steered virtual character (e.g. join rotations, morphs). A python script is used to synchronize the two character representations. An implementation of the renderer is provided in Panda3D.

[8]In this comparison we have made use of the SmartBody version as obtained through the SmartBody SVN at July 1st 2012, EMBR release 0.5.2 and the articles describing the system [11, 13], the Greta version that was freely available online at July 1st 2012, the latest version of ACE at July 1st 2012, version 0.9 beta of Elckerlyc and the latest version of AsapRealizer at July 1st 2012.

[9]`http://www.unreal.com/`, `http://www.panda3d.org/`, `http://www.emergent.net/`, `http://www.valvesoftware.com/`, and `http://www.ogre3d.org/`

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.

AsapRealizer controls animation through the Embodiment interfaces discussed in Section 4.3. New renderers are supported through implementing two interfaces: the embodiment interface through which one controls a joint hierarchy, plus a loader class for instantiating it. AsapRealizer can control the skeleton in any new render environment as soon as the library is added to the class path. We have, so far, made several implementations: one for our own OpenGL renderer; a simple custom XML based TCP/IP protocol that allows AsapRealizer to control avatars running in the SUIT render environment of Re-Lion[10]; and a proof-of-concept implementation that allows AsapRealizer to control avatars in the Ogre engine. The latter uses the Thrift remote procedure call (RPC) framework [7] to handle its communication with the renderer. Unlike the BoneBus library, Thrift 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). ACE follows a similar design strategy.

5.2. Integration with New and Existing Text-To-Speech Systems

SmartBody and ACE allow one to replace the Text-To-Speech (TTS) system without recompilation. To do this one needs to implement a plugin-module (e.g. as dll) that links the desired text to speech system to an interface standardized for the realizer. AsapRealizer provides integration with Text-To-Speech systems in a similar manner. Additionaly, it provides the functionality to hook up multiple (might be instances of the same, configured diferently) TTS systems, and to support the use of multiple TTS speech markup languages in alternation for the same loaded virtual human (e.g. MaryTTS script, SSML, MS SAPI).

5.3. BML Transport Wiring

When a VH is to be integrated into an application, it may be necessary to be able to control the VH by sending BML from another programming language, a different Operating System, or a remote machine.

[10]`http://www.re-lion.com`

SmartBody offers integration with the Active MQ[11] 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 [14] messaging frameworks to achieve this; Greta additionally offers integration with Psyclone.[12]

In AsapRealizer, a crucial requirement was that the BML transport could easily be adapted. To this end, we took a different design philosophy for BML transport than the realizers mentioned above. Rather than relying on one (or few) BML transport mechanisms or middleware systems, we argue that BML transport is not a responsibility of the core realizer itself. Therefore we provide a clean and simple interface (in Java) in which BML strings can be send to the realizer and feedback listeners can be registered. Adapters and pipes in seperate modules are used to compose more intricate BML transport mechanisms. Current implementations of such modules include adapters for the SEMAINE system, for ActiveMQ and a simple direct TCP/IP connection and pipes that allow logging and throttling for multithreaded execution. Our design philosophy is similar to that used to compose complex functionality out of the composition of several simple Unix programs that are connected with simple standard interfaces (e.g. text through the stdout) [15] and to Alistair Cockburn's Hexagonal architecture.[13]

5.4. BML to Output Mapping

All realizers provide functionality to map BML behaviors to scripted units (e.g. keyframe animations or predefined animation scripts). SmartBody provides a configuration file in which one can set up a one to one mapping from gesture lexeme to keyframe animation file. EMBR and Greta convert the BML behavior into a query that is used to search their behavior lexicons; adaptations in existing animations in the lexicon and additions to the lexicon are thus automatically handled in these realizers. In ACE, animations are constructed dynamically on the basis constraints specified in the MURML script language. In addition to being able to run scripted animation, these realizers also provide several procedural animation systems that are hardcoded in the realizer, including gaze systems, locomotion systems or pointing systems. However, none of the realizers mentioned above allows one

[11]`http://activemq.apache.org/`

[12]`http://www.cmlabs.com/psyclone/`

[13]`http://alistair.cockburn.us/Hexagonal+architecture`

to add a new procedural (e.g. locomotion, pointing, gaze) output or change the existing one, without recompiling their core system.

AsapRealizer contributes the ability to specify the mapping of BML to procedural output units *without* requiring modifications to AsapRealizer's source code. This flexibility is offered by AsapRealizer's binding (see Section 4.2). Like the lexicons of Greta and EMBR, the binding can be queried by BML. In addition to that, the binding can map BML parameters to e.g. animation parameters and can provide default parameter values. The latter allows reuse of e.g. an animation for different BML behaviors. For example: a keyframe animation of a left hand gesture can also be as a right hand gesture, by setting its mirror parameter to true.

5.5. Output Modality

In SmartBody, EMBR, Greta and ACE it is not possible to change the modality of a BML behavior (e.g. a virtual human vs a robotic head vs a cartoon head, text vs TTS) without recompilation of the realizer. AsapRealizer provides Engine and Embodiment abstractions to allow this. This architectural feature is discussen in Sections 4.3 and 4.5.

5.6. Lipsync Algorithm

In SmartBody, EMBR, Greta, ACE and Elckerlyc, the lipsync algorithm and output modality are hardcoded in the realizer. In AsapRealizer, lipsync module(s) can be registered to the SpeechEngine, without recompilation of the SpeechEngine or AsapRealizer's core. This allows one to both apply different lipsync strategies (e.g. new strategies that allow co-articulation) and to easily apply lipsync on different embodiments (e.g. a robot or a 2D cartoon character). AsapRealizer's lipsync strategy is dicussed in detail in Section 4.8.

5.7. Available Behavior Types and Extensibility

There are many different paradigms for the generation of specific behaviors for avatars (see, e.g., [16] for a survey of animation techniques). The choice which paradigm is the most suitable in a given situation depends (also) on the application. A BML Realizer should therefore support as many behavior types as possible, and should preferebly be easy to extend with new types.

Smartbody uses 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 [17] for facial animation.

AsapRealizer allows all of the above, and adds physically simulated animation behaviors and sound effects (one can specify sound files to be played in synchronisation to other behaviors, through a custom SoundEngine with a sound behavior BML extension). More importantly, we contribute the ability to add custom behavior types and output modalities *without* requiring modifications to AsapRealizer's source code, as described in Sections 4.4 and 4.5.

6. Discussion

We have discussed how AsapRealizer can be tailored to the needs of specific applications, without requiring invasive modifications to AsapRealizer itself. AsapRealizer'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. For example, a group of educational technology researchers succeeded in sending BML from their tutoring application to AsapRealizer, embedding our virtual human as embodied tutor in their educational software, doing a series of user experiments with this setup. In our own experiments, the logging port allowed us to easily record all communication with AsapRealizer for user experiments, by simply changing the wiring between the behavior planner and AsapRealizer. 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 AsapRealizer, but also test Realizers designed by other research groups (by writing an adaptor from the BMLRealizerPort to their input and output channels), as discussed in [18]. AsapRealizer's ability to add new modalities has allowed us to hook it up with the Nabaztag rabbit (see also Section 4.5) 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 AsapRealizer's source code.[14] An Embodiment and Engine implementation for a humanoid robot head was implemented by another research group, allowing them to steer the head and gaze behavior of their robot using BML requests that are realized by AsapRealizer.

AsapRealizer is compliant with BML version 1.0. Switching from the Draft 1.0 version of BML was mostly a matter of updating the SAIBA BML parsing packages[15] and updating the XML descriptions for the various Bindings. Minor changes to the standard will require no more than that; slightly more extensive changes require modifications of the scheduler and the engines, but those are often still local to a specific engine. The BML Realizer Tester framework mentioned above helps in smoothly moving to new versions of the BML standard.

AsapRealizer's extensibility is mainly achieved by a very flexible initialization stage. In this initialization stage, a desired setup of the AsapRealizer is constructed by combining and configuring different components that are provided by AsapRealizer'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.

7. Acknowledgements

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

[14]See `http://elckerlyc.sourceforge.net/` for screenshots and movies.
[15]available from `http://sourceforge.net/projects/saibabml/`

References

[1] H. van Welbergen, D. Reidsma, Z. M. Ruttkay, J. Zwiers, Elckerlyc: A BML realizer for continuous, multimodal interaction with a virtual human, Journal on Multimodal User Interfaces 3 (2010) 271–284.

[2] H. van Welbergen, D. Reidsma, S. Kopp, An incremental multimodal realizer for behavior co-articulation and coordination, in: Y. Nakano, M. Neff, A. Paiva, M. A. Walker (Eds.), IVA, volume 7502 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 175–188.

[3] S. Kopp, I. Wachsmuth, Synthesizing multimodal utterances for conversational agents, Computer Animation and Virtual Worlds 15 (2004) 39–52.

[4] S. Kopp, B. Krenn, S. Marsella, A. N. Marshall, C. Pelachaud, H. Pirker, K. R. Thórisson, H. H. Vilhjálmsson, Towards a common framework for multimodal generation: The behavior markup language, in: IVA, pp. 205–217.

[5] D. Reidsma, H. van Welbergen, J. Zwiers, Multimodal plan representation for adaptable bml scheduling, in: H. Vilhjálmsson, S. Kopp, S. Marsella, K. Thórisson (Eds.), Intelligent Virtual Agents - 11th International Conference, IVA 2011, Reykjavik, Iceland, September 15-17, 2011. Proceedings, volume 6895 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2011, pp. 296–308.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Adisson-Wesley, 1995.

[7] M. Slee, A. Agarwal, M. Kwiatkowski, Thrift: Scalable cross-language services implementation, 2007.

[8] K. Balci, Xface: MPEG-4 based open source toolkit for 3d facial animation, in: AVI04, Working Conference on Advanced Visual Interfaces.

[9] R. Klaassen, J. Hendrix, D. Reidsma, R. op den Akker, Elckerlyc goes mobile: Enabling technology for ecas in mobile applications, in: UBICOMM 2012, The Sixth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, pp. 41–47.

[10] M. Thiebaux, A. N. Marshall, S. Marsella, M. Kallmann, Smartbody: Behavior realization for embodied conversational agents, in: AAMAS, pp. 151–158.

[11] A. Heloir, M. Kipp, Real-time animation of interactive agents: Specification and realization, Applied Artificial Intelligence 24 (2010) 510–529.

[12] M. Mancini, R. Niewiadomski, E. Bevacqua, C. Pelachaud, Greta: a SAIBA compliant ECA system, in: Agents Conversationnels Animés.

[13] M. Kipp, A. Heloir, M. Schröder, P. Gebhard, Realizing multimodal behavior: Closing the gap between behavior planning and embodied agent presentation, in: J. Allbeck, N. Badler, T. W. Bickmore, C. Pelachaud, A. Safonova (Eds.), Proceedings of the 10th International Conference on Intelligent Virtual Agents, volume 6356 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 57– 63.

[14] M. Schröder, The SEMAINE API: Towards a standards-based framework for building emotion-oriented systems, Advances in Human-Computer Interaction (2010).

[15] E. S. Raymond, The Art of UNIX Programming, Addison-Wesley, 2003.

[16] H. van Welbergen, B. J. H. van Basten, A. Egges, Z. M. Ruttkay, M. H. Overmars, Real time animation of virtual humans: A trade-off between naturalness and control, Computer Graphics Forum 29 (2010) 2530–2554.

[17] P. Ekman, W. Friesen, Facial Action Coding System: A Technique for the Measurement of Facial Movement., Consulting Psychologists Press, Palo Alto, 1978.

[18] H. van Welbergen, Y. Xu, M. Thiébaux, W.-W. Feng, J. Fu, D. Reidsma, A. Shapiro, Demonstrating and testing the bml compliance of bml realizers, in: H. H. Vilhjálmsson, S. Kopp, S. Marsella, K. R. Thórisson (Eds.), Intelligent Virtual Agents - 11th International Conference, IVA 2011, Reykjavik, Iceland, September 15-17, 2011. Proceedings, volume 6895 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 269–281.

Appendix A: Engines and Embodiments

Table 2: Overview of Engines available in AsapRealizer, the BML behaviors they can deal with, the Plan Units they use for this, and the Embodiments (and their control primitives) controlled by the Engine.

TTSEngine	
Core BML:	`speech`
BML extensions:	Dynamically set by binding. E.g: SSML, MS SAPI, MaryXML
Plan Units:	TimedTTSUnit (impl: TimedWavTTSUnit, TimedDirectTTSUnit)
Embodiment interface:	TTSGenerator
Implementations:	AndroidTTSGenerator, MaryTTSGenerator, SAPI5TTSGenerator
TextEngine	
Core BML:	`speech`
Plan Units:	TimedTextSpeechUnit
Embodiment interface:	TextOutput
Control primitives:	text
Implementations:	JLabelTextOutput, StdoutTextOutput
AnimationEngine	
Core BML:	`head, gaze, gesture, posture, postureShift, pointing`
BML extensions:	`procanimation, controller, keyframe, noise, murmlgesture`
Plan Units:	TimedAnimationUnit (impl: PointingUnit, ProcAnimationUnit, GestureUnit, NoiseUnit, PhysicalControllerUnit, KeyframeUnit, MURMLUnit, GazeUnit, PostureUnit)
Embodiment interface:	SkeletonEmbodiment
Control primitives:	joint rotation, translation
Implementations:	HmiRenderBodyEmbodiment, RelionEmbodiment
Embodiment interface:	PhysicalEmbodiment

Table 2 – *Continued from previous page*

Control primitives:	joint torque, root force
Implementations:	OdePhysicalEmbodiment
FaceEngine	
Core BML:	`faceFacs, faceLexeme`
BML extensions:	`murmlface, facemorph`
Plan Units:	TimedFaceUnit(impl: AUUnit, FACSUnit, MorphUnit, PlutchikUnit)
Embodiment interface:	MorphEmbodiment
Control primitives:	morph targets
Implementations:	FaceController
Embodiment interface:	MPEG4Embodiment
Control primitives:	MPEG-4 FAPS
Implementations:	FaceController, XFaceController
AudioEngine	
BML extensions:	`audiofile`
Embodiment interface:	SoundManager
Control primitives:	audio
Implementations:	ClipSoundManager (java default), JoalSoundManager (openal)
NabaztagEngine	
BML extensions:	`moveearto, wiggleear`
Plan Units:	MoveEarToNU, WiggleEarNU
Embodiment interface:	NabaztagEmbodiment
Control primitives:	earposition
PictureEngine	
Core BML:	`faceLexeme, gesture`
BML extensions:	`setImage, addImage, addAnimationXML, addAnimationDir`
Plan Units:	TimedPictureUnit(impl: SetImagePU, AddImagePU, AddAnimationPU, AddXMLPU)
Embodiment interface:	PictureEmbodiment
Control primitives:	add/remove/replace image at layer x
Implementations:	JFramePictureEmbodiment, AndroidPictureEmbodiment

Table 2 – *Continued from previous page*

FlobiEngine	
Core BML:	`gaze, head, faceFacs`
Plan Units:	Facs, HeadOrientation, HeadRotation
Embodiment interface:	XS2Output
Control primitives:	NAORSB calls (nao is a misnomer here, same library is used for both Flobi and Nao)
NaoEngine	
Core BML:	`locomotion, head, faceFacs`
Plan Units:	Locomotion, HeadOrientation, HeadRotation
Embodiment interface:	XS2Output
Control primitives:	NAORSB calls
LiveMocapEngine	
BML extensions:	`remoteFaceFACS, remoteHead`
Plan Units:	LiveMocapTMU(impl: RemoteHeadTMU, RemoteFaceFACSTMU)
Embodiment interface:	a PlanUnit specific (Sensor,Embodiment) pair
Implementations:	EulerInput-¿EulerHeadEmbodiment, FACSFaceInput-¿FACSFaceEmbodiment