Noname manuscript No.
(will be inserted by the editor)

Multimodal Plan Representation for Adaptable BML Scheduling

Herwin van Welbergen · Dennis Reidsma · Job Zwiers

Received: date / Accepted: date

Abstract Natural human interaction is characterized by interpersonal coordination: interlocutors converge in their speech rates, smoothly switch speaking turns with virtually no delay, provide their interlocutors with verbal and nonverbal backchannel feedback, wait for and react to such feedback, execute physical tasks in tight synchrony, etc. If virtual humans are to achieve such interpersonal coordination they require very flexible behavior plans that are adjustable on-the-fly. In this paper we discuss how such plans are represented, maintained and constructed in our BML realizer Elckerlyc. We argue that behavior scheduling for Virtual Humans can be viewed as a constraint satisfaction problem, and show how Elckerlyc uses this view in its flexible behavior plan representation that allows one to make on-the-fly adjustments to behaviors while keeping the specified constraints between them intact.

Keywords Virtual Humans · Behavior Markup Language · SAIBA · multimodal plan representation · interpersonal coordination

1 Introduction

A virtual human uses verbal and nonverbal behavior to express (communicative) intentions, in a dialog or other interaction context. This should not happen in monolotihic series of monologues: a virtual human need to be able to

This research has been supported by the GATE project, funded by the Dutch Organization for Scientific Research (NWO), and by the GATE KTP project. This paper is an extended and revised version of a paper presented at IVA 2011.

H. van Welbergen
Sociable Agents Group, CITEC, University of Bielefeld
E-mail: hvanwelbergen@techfak.uni-bielefeld.de

H. van Welbergen · D. Reidsma · J. Zwiers
Human Media Interaction Group, University of Twente
E-mail: {d.reidsma,j.zwiers}@utwente.nl

on-the-fly adapt their ongoing behavior. They need to add actions in the middle of a sentence (e.g., look up briefly at a passer-by), interrupt themselves, change the timing of their speech and gestures to accomodate behavior of the interlocutor such as feedback and interruptions. Goodwin describes an example of such adjustments in human-human conversation: when a listener utters an *assesment* feedback, the speaker, upon recognizing this, will slightly delay subsequent speech (e.g. by an inhalation or production of a filler) until the listener has completed his assessment (Goodwin 1986). A virtual human might also have to closely coordinate their movements with those of a human partner while performing a joint *physical* task. For example, when a virtual sports coach is performing an exercise along with the user, it needs to continually update the exact timing with which it performs the movements in order to stay synchronized with the user. Our aim is to build virtual humans with such capabilities for human-like interpersonal coordination. For this, we need to be able to make on-the-fly adjustments to the behavior being displayed (Kopp 2010; Nijholt et al 2008).

This paper describes three important contributions towards achieving this flexibility.

(1) We describe how the addition of verbal and nonverbal behavior to an ongoing motor plan can be viewed as a constraint satisfaction problem: a collection of behaviors, with constraints on their possible timing, is added to a flexible (motor) plan in such a way that these constraints are met, and existing constraints in the plan are retained. Several algorithms (including constraint programming/constraint optimization) can be used to solve this constraint satisfaction problem.

The focus of our paper is on the *constraint representation* rather than on algorithms to solve the constraint satisfaction problem – for the latter we make use of an improved version of the constraint solving strategy used in the SmartBody realizer and illustrate how other strategies might be incorperated in our work. We show why it is important that these constraints are represented *explicitly* in some way in the motor plan, for maintenance of (adherence to) the set of constraints in case of on-the-fly adaptation or incremental extension of the plan.[1]

(2) We introduce our implementation of a novel and flexible intermediate motor plan representation in which we can easily make the required kinds of on-the-fly adaptations to the behavior of the virtual human.

(3) Finally, we discuss practical examples of how we use these new capabilities, implemented in Elckerlyc (van Welbergen et al 2010), to do a number of things that would not have been possible (or at least: more difficult to achieve) without the above.

[1] We use this "constraint problem" view to develop a novel flexible motor plan representation, but its value is actually broader than that. It can also contribute to clarifying standards for multimodal behavior generation, validating behavior realizers and/or scripts, etcetera. Having formal constraints also means that you can (partially) express the expected end result for a BML expression independently from the system used to generate the virtual human's behavior, facilitating systematic testing and validation (van Welbergen et al 2011).

2 Behavior planning in SAIBA

The SAIBA framework (Kopp et al 2006) provides a good starting point for designing interactive virtual humans. Fig. 1 shows a slightly elaborated version of the SAIBA reference architecture for behavior generation. The *Intent Planner* is responsible for generating the higher level communicative intentions of the virtual human, in a dialog or any other interaction setting. Example intentions are to ask the user for their name, to explain a mathematics exercise, to compliment the user on their new hair cut, etcetera.

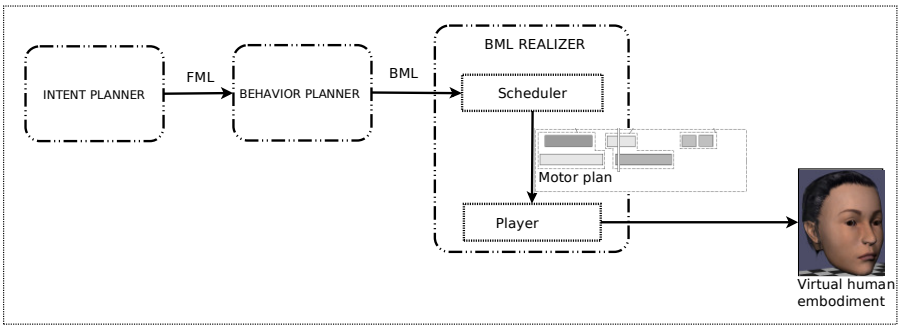


Fig. 1: Simplified overview of the SAIBA virtual human behavior generation pipeline. Communicative intent is translated to a multimodal behavior plan specified in BML. This BML is performed on the embodiment of the virtual human in two steps, by the BML Realizer. The scheduler takes the BML and adapts/extends the current motor plan. The player executes the motor plan on the embodiment of the virtual human through sound and movement.

The *Behavior Planner* specifies the verbal and nonverbal behavior that should be used to express these intentions. The type of behavior, and the constraints on its timing, are specified using the Behavior Markup Language (BML). A single BML block typically contains a number of behavior elements; alignment and timing are specified in reference to *sync points* such as the start or end of a behavior. Fig. 2 shows an example BML block and the standard sync points of a BML gesture behavior. A stream of BML blocks is sent from the behavior planner to the BML realizer.

The *BML Realizer*, finally, is responsible for displaying the content of the BML block on the embodiment of the virtual human, using sound and motion. The BML realizer should execute the behaviors in such a way that the time constraints specified in the BML blocks are satisfied. Realizer implementations typically handle this by separating the BML scheduling process from the playback process. As can be seen in Fig. 1, the scheduling process generally converts the BML blocks to a motor plan that can be directly displayed by the playback process on the embodiment of the virtual human.

Fig. 3 shows how the motor plan is extended with new elements, based on BML that has been sent by the behavior planner to the BML realizer. The motor plan representation forms an intermediate level between the multimodal

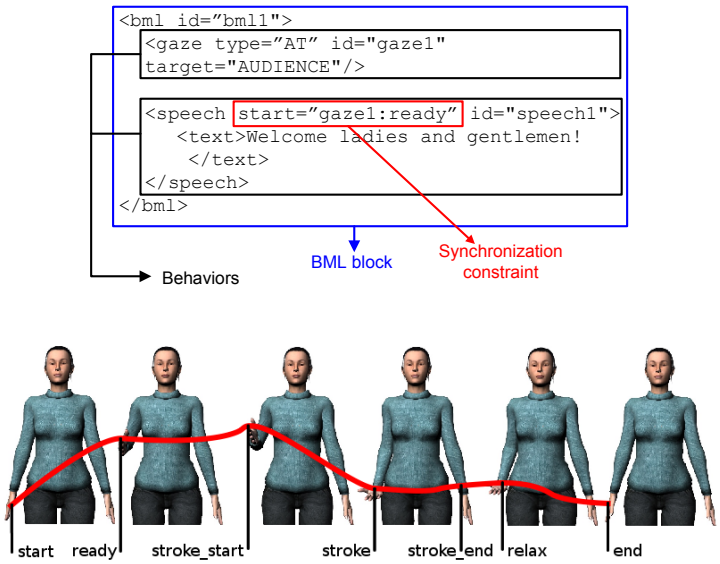


Fig. 2: BML is used to specify the required behavior of the virtual human, and the various alignments between the single behaviors. This figure shows an example BML script and the standard sync points of a BML gesture behavior.

behavior plan (BML) and the surface realization on the embodiment of the virtual human (body movement, speech audio, facial movement, etcetera). In most BML Realizers scheduling the stream of BML blocks results in a *rigid* motor plan. Once scheduled, the plan cannot be modified very well – at best, a Realizer allows one to drop (a subset of) the current plan and replace it with a new plan. The more flexible plan representation that is introduced later in this paper allows one to interrupt behaviors, change the timing of synchronisation points, add additional behaviors, and change the parameterization of behaviors on-the-fly while keeping the constraints intact. This makes it eminently suitable for VH applications in which a tightly coordinated interaction between user and VH is required.

3 BML Scheduling as a Constraint Problem

BML expressions specify behaviors to be realized by a BML Realizer, and their timing and alignment. Fig. 3 already showed how the scheduling process creates and maintains the intermediate multimodal motor plan that will be displayed on the virtual human's embodiment at playback time. In this section we look in more detail at the various types of constraints set to the motor plan by the BML, and describe how scheduling can be thought of as constructing a motor plan that adheres to these constraints.

A new BML block u is sent to the scheduler at time ct (indicated by the vertical white bar in Fig. 3). The block u specifies new behaviors $\mathbf{b}$ with sync

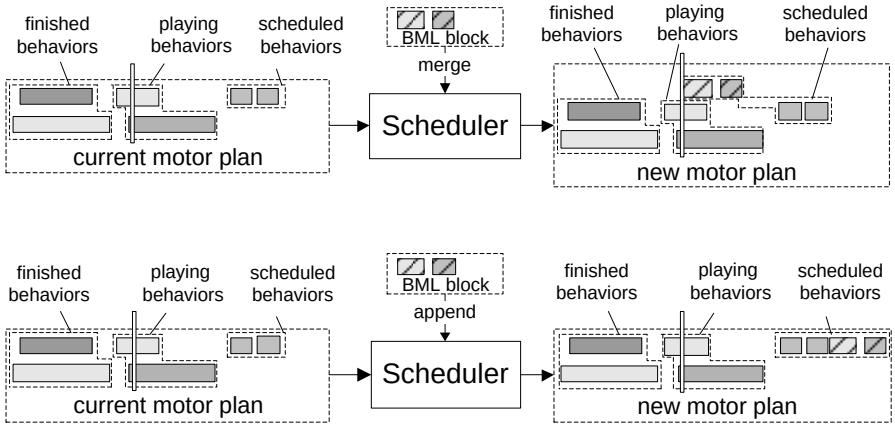


Fig. 3: The scheduling process that transforms a stream of BML into a motor plan. The white bar indicates the current time ct. The new BML block u defines how the currently playing and planned behaviors are updated and which new behaviors are inserted, using a `composition` attribute. `merge` (top) specifies that the behaviors in the BML block are to be started at the current time. `append` (bottom) indicates that the behaviors in the BML block are to be inserted after all behaviors in the current plan.

points $\mathbf{s}$ (such as start, stroke, or end) and their alignment. The scheduling process of a realizer updates the current motor plan on the basis of u.

A scheduling function $f : \mathbf{s} \rightarrow \mathbf{t}$ maps sync points s to global time t. Another scheduling function $blockstart : \mathbf{u} \rightarrow \mathbf{t}$ maps blocks u to their global start time t. The goal of scheduling is to find the values of $f(s)$ for all sync points in all behaviors in a new block u as well as the value of $blockstart(u)$, in such a way that all constraints are satisfied. Because most BML blocks are underspecified, schedulers have a lot of freedom to solve this in such a way as to obtain nice and natural looking animations.

The behaviors are added to the motor plan subject to a set of timing constraints $\mathbf{c}$. Firstly, there are the constraints that are explicitly defined in the BML block specification. Secondly, there are certain implicit constraints that hold for any BML block (e.g., behaviors should start before they end). Thirdly, a specific realizer can impose additional constraints upon the scheduling, motivated by biological capabilities of the virtual human it steers. Technical limitations (e.g. inflexible timing of Text To Speech Systems) might further constrain the timing of the behavior plan. Finally, Block Level Constraints, as specified by the `composition` attribute in the BML block, define the relation between the start of the to-be-scheduled BML block and the behaviors already present in the current motor plan (see the difference between the two examples in Fig. 3). The five types of constraints are described in more detail below using BML Example 1. Appendix A contains a more formal and detailed treatment.

BML Example 1 Basic BML example used to explain the constraints.

```
<bml id="bml1" composition="append">
    <speech id="speech1">
      <text>As you can see on <sync id="s1"> this painting, ...</text>
    </speech>
    <gesture id="point1" type="POINT" target="painting1"
             stroke="speech1:s1+0.5"/>
    <face id="f1" start="0" end="4" type="LEXICALIZED" lexeme="smile"/>
</bml>
```

3.1 Explicit Constraints

Explicit time constraints are specified directly in the BML expression, as a time relation between *sync references*. A sync reference consists of either a time value in seconds, denoting an offset from the start of the BML block, or a sync point of one of the behaviors and an offset (may be 0). BML defines two types of time relations:

- *before/after*: sync reference a occurs before (or after) sync reference b.
- *at*: sync references a and b occur at the same time.

In Example 1, the `stroke` of the gesture is constrained to be 0.5 seconds after `s1` of the speech. The sync references involved in this constraint are expressed as $[[[bml1, speech1], s1], 0.5]$ and $[[[bml1, point1], stroke], 0]$. Given the notations and definitions from Appendix A, the constraint on these sync references comes out as

$$c_r = [[[bml1, speech1], s1], [[bml1, point1], stroke], -0.5] \quad (1)$$

Explicit constraints typically express the multimodal timing of behavior. They also provide the Behavior Planner with the ability to define those constraints on a behavior that maintain the intended meaning of a behavior.

3.2 Implicit Constraints

Apart from the explicit constraints defined in the BML block, several implicit constraints act upon f:

1. Sync points may not occur before the block in which they occur is started.
2. Behaviors should have a nonzero duration.
3. The default BML sync points of each behavior (for gestures: `start`, `ready`, `stroke_start`, `stroke`, `stroke_end`, `relax`, `end`) must stay in that order.

For example, the first point implies that the time assigned to the syncpoints must be greater than the start time of the example block, even though no start time was specified in the BML block for those two behaviors. More formally: for all syncpoints $s \in \mathbf{s}$ of behaviors $speech1$ and $point1$ in block $bml1$, $f(s) \geq blockstart(bml1)$

In addition to the constraints mentioned above, a set of implicit cluster constraints enforces that there is no 'unnecessary whitespace' between behaviors (see Appendix A.3.2 for a more rigourous treatement). That is, each behavior, as well as each block, is supposed to start as early as possible, as long as it satisfies all other constraints.

3.3 Biomechanical Constraints

Realizers might impose additional biomechanical constraints that are typically behavior specific. A Realizer might, e.g., forbid solutions that require a VH to gesture at speeds beyond its physical ability.

3.4 Technical Constraints

Other constraints are due to a technical limitations of current behavior realization techniques. For example, most Text-To-Speech systems do not allow one to make detailed changes to the timing of the generated speech. Therefore, realizers typically forbid scheduling solutions that require the stretching of speech behaviors beyond the default timing provided by the TTS system.

3.5 Block Level Constraints

The `composition` attribute associated with a BML Block (see also Fig. 3) defines constraints on the start of the block in relation to the set of current behaviors in the motor plan and to the current global time ct. BML defines the following composition attributes:

1. `merge`: start the block at ct.
2. `replace`: remove all behaviors from the current plan, start the block at ct.
3. `append`: start the block as soon as possible after all behaviors in the current plan have finished (but not earlier than ct).

In Example 1, the composition is `append` so the start time of the block must be greater than the end time of everything that is currently in the motor plan, as well as greater than ct. Clearly, these equations can also be rewritten in terms of the start time of all behaviors $b \in \mathbf{b}$ in BML block $bml1$, constrained relative to the end time of all behaviors already in the current motor plan. This is also explained in Appendix A.

3.6 Additional Behavior Plan Constraints In Elckerlyc

We have defined extensions to BML that allows us to specify an additional block level constraint: in addition to the `merge` and `append` block composition attributes, Elckerlyc provides the `append-after(X)` attribute. This starts a

BML block directly after a selected set of behaviors in the current behavior plan (those from all blocks in **X**) are finished. The block level constraints for this composition attribute can, again, be rewritten into constraints on only the *behaviors* in the current plan and in the new block. A more formal and extensive treatment of Elckerlyc's BML block constraint extensions can be found in Appendix A.5.

3.7 Meaning retaining Constraints

Behaviors might be executed in ways that that are biomechanically plausible, but that validate the meaning intended by the behavior planner. In our view, a BML Realizer should not be responsible for the maintainance of intended meaning and intended meaning should not be expressed in BML. Instead, the behavior planner should express the time constraints it requires to retain meaning explicitly in BML (perhaps using a BML behavior library in which such constraints are annotated per intention). However, the exact semantics of BML 1.0 are still open and it has been suggested to map BML behavior elements to gesture repertoire elements in which meaning retaince constraints are annotated. Such a behavior construction technique is compatible with our solver and constraint maintenace mechanisms, as meaning retaining constraints can simple be added to the constraint representation.

4 Existing BML Scheduling Solutions

In this section we describe the scheduling solutions implemented in the BML Realizers SmartBody and EMBR. Both EMBR and SmartBody apply top down scheduling, resulting in a rigid behavior plan. More flexible behavior scheduling (albeit not within the SAIBA framework) was previously achieved in the ACE system, for the specific application of gesture co-articulation.

4.1 Top down, rigid scheduling

EMBR (Heloir and Kipp 2010; Kipp et al 2010) uses a constraint optimization technique to solve the scheduling problem. The EMBR scheduler first solves the absolute value of all BML sync points in speech. A timing constraint solver then solves for the timing of the remaining nonverbal behaviors. Synchronization constraints might require the stretching or shortening of behavior phases as compared to the defaults given in the behavior lexicon. The constraint solver uses the sum of 'errors' (in seconds) of the stretch over all behaviors as its cost function. It thus finds solutions in which the overall stretch is minimized. The EMBR scheduler can schedule BML blocks containing before and after constraints, and favors solutions that result in more natural behavior (for EMBR's measure of the naturalness: minimal overall behavior stretching).

SmartBody (Thiebaux et al 2008) uses a very fast custom scheduler that does not use constraint optimization techiques. SmartBody's scheduling algorithm solves the constraints in the following way. It processes the behaviors in a BML block one by one, in the order in which they appear (syntactically) in the block. Given the next behavior, it assigns an absolute timing to its sync points so that they adhere to all timing constraints posed by its (syntactic) predecessors in the BML block and to any absolute time constraints (offset from the start time of the block). Once the a behavior is processed, its timing is fixed; if a subsequent behavior must be aligned to it, those constraints are solved by manipulating that subsequent behavior. If two time constraints on a behavior require certain phases of that behavior to be stretched or skewed, the scheduler achieves this by stretching or skewing the behavior uniformly, to avoid discontinuities in animation speed. SmartBody's scheduling mechanism can result in some time constraints being scheduled into the past (that is, before the start of the BML block). A final normalization pass is performed in which, where needed, connected clusters of behaviors are shifted forward in time to fix this. SmartBody cannot handle before/after constraints yet, but does comply with all explicit constraints and implicit constraints that do not concern before and after constraints.

Because the SmartBody scheduling algorithm schedules behaviors in the order in which they syntactically appeared in the BML block, this ordering, which should not have a semantic effect, can actually influence the scheduling solution. At worst, this may lead to situations in which BML cannot be scheduled in one order while it can be in another. For example, the BML block in Example 2(a) cannot be scheduled because the timing of the `nod1` is determined first, and the scheduler attempts to retime `speech1` to adhere to this timing. Most speech synthesis systems, including the one used in SmartBody, forbid such retiming. If the behavior order is changed, as in Example 2(b), then `speech1` is scheduled first, and `nod1` will adhere to the timing imposed by `speech1`. That being said, the SmartBody scheduling algorithm is easy to implement and provides rapid scheduling. In practice, most BML scripts are simple and the SmartBody scheduler will find a reasonable scheduling solution for such scripts.

Two motor plan properties are important for flexible plan adaptation: Firstly, one must maintain a grounding from units in the motor plan to the BML expressions that resulted in them being added to the plan. This means that even after the motor plan is constructed, it is possible to refer to (and modify or remove) units of this plan using their original BML identifiers. Secondly, the constraints that act upon the plan must be still represented after it has been scheduled, so that modifications to the plan can be made that do not invalidate the constraints specified in BML. Both EMBR's and SmartBody's scheduling approaches are applied in one-shot fashion: scheduling resolves BML behaviors and constraints into a plan describing the absolute timing of to be executed motor units (e.g. audio files for speech and keyframe animation for gestures). While their motor plan (to some extent) retains a grounding of units in the motor plan to their matching BML behaviors (this is

BML Example 2 Two BML scripts demonstrating SmartBody's order dependent scheduling solution.

(a) BML script that cannot be scheduled using the SmartBody scheduling algorithm.

```
<bml id="bml1">
  <head id="nod1" action="ROTATION" rotation="NOD" start="speech1:start"
   end="speech1:sync1"/>
  <speech id="speech1">
    <text>Yes,<sync id="sync1"> that was great.</text>
  </speech>
</bml>
```

(b) Equivalent BML script that *can* be scheduled using the SmartBody scheduling algorithm.

```
<bml id="bml1">
  <speech id="speech1">
    <text>Yes,<sync id="sync1"> that was great.</text>
  </speech>
  <head id="nod1" action="ROTATION" rotation="NOD"
    start="speech1:start" end="speech1:sync1"/>
</bml>
```

also required for behavior progress feedback), they do not represent the BML constraints in the motor plan. Therefore their approach lead to a rigid motor plan that cannot easily be modified.

4.2 Combining top-down scheduling with last-minute bottom up scheduling

Both EMBR and SmartBody employ top-down scheduling mechanisms that are steered using a central scheduler with a single scheduling step that directly translate BML into a rigid motor plan. However, some information on the timing of behavior is not readily available at the start of the BML block it is in. For example, the duration of the preparation phase of a gesture is dependent on the position of the hand at the start of the gesture, which typically only known close to the actual start time of the gesture. For other behaviors – such as robot gesture – it is hard to predict the timing precisely beforehand. To achieve synchronization for such behaviors, it makes sense to adapt their timing flexibly be adapted in a bottom up (so from the modules executing the behavior, rather from a central scheduler), last-minute manner.

A combination of top down and bottom up scheduling was previously employed in the ACE system (Kopp and Wachsmuth 2004). ACE executes multimodal behavior using successive 'chunks' containing one tone unit in speech and a co-expressive gesture phrase. The gesture phrase is aligned to the tone unit in such a way that the stroke phase of the gesture starts before (in ACE this is 0.3s or one syllable) the affiliate in speech. Because of technical limitations of the text-to-speech system employed in ACE, the duration of the tone unit is completely fixed. A top down scheduler first resolves as much of the

timing and shape of the chunk as possible: it synthesizes the tone unit, selects a lexicalized gesture template, allocates body parts, expands abstract movement constraints and resolves deictic references. A second top-down scheduling step starts as soon as the chunk has to be started: now the scheduler needs to determines whether to start with the gesture or with the tone unit. If the chunk starts with the gesture, the duration of the preparation phase of the gesture is determined based upon the current hand position, and the start time of the tone unit is determined in such a way that the synchronization constraint between the gesture's stroke and the tone-unit's affiliate in speech is satisfied. The timing of all behaviors is then determined and no further time adaptations are required. Typically however, the chunk is to start with the tone unit in speech. The second top-down scheduling step then determines the timing of the tone unit, and provides a first prediction of the start time of the gesture, based upon the current position of the hand. Predictions for the start time of the gesture are made continuously while the chunk is being played, by the process playing the gesture animations (thus in a bottom-up fashion). As soon as the predictions are at or past the current time, the gesture start time is commited and the gesture is started. In Section 6.2 we show how such bottom up scheduling can be used with Elckerlyc's plan representation and how we can generalize it to provide synchronization strategies between multiple modalities that all provide flexible re-timing.

5 Scheduling and Plan Representation in Elckerlyc

Our BML Realizer "Elckerlyc" (van Welbergen et al 2010) was designed specifically for highly adaptive behavior generation. Its multimodal behavior plan can continually be updated: the timing of certain synchronisation points can be adjusted, ongoing behaviors can be interrupted using special "interrupt behaviors", behaviors can be added, and the parametrisation of ongoing behaviors can be changed.[2] In order to achieve this flexibility, Elckerlyc needs not only to be able to schedule BML specifications into a motor plan that describes the surface realization of the behaviors, but also to maintain information about how these surface realizations relate to the original BML specification. This allows the scheduler to figure out which surface realizations need to be changed, when changes to BML behaviors are requested.

In this section we describe our implementation of Elckerlyc's novel motor plan representation and scheduling approach that achieves this.

5.1 Scheduling in Elckerlyc

The Elckerlyc scheduling architecture uses an interplay between different unimodal Engines that are specialized in planning the motor behaviors for one specific modality (e.g. Speech Engine, Animation Engine, see also Fig. 4).

[2] The mechanisms for specifying these changes are described in (Zwiers et al 2011).

Elckerlyc's motor plan representation (cf. Fig.1) is actually distributed over separate unimodal plans in each Engine. These unimodal plans contain Timed Plan Units, describing in detail the control of an embodiment (i.e., movement or sound) required to display behaviors from the BML behavior plan. The Peg Board, discussed in detail later, is used to maintain information about absolute timing constraints for Timed Plan Units as well as timing constraints between Timed Plan Units in different modalities.

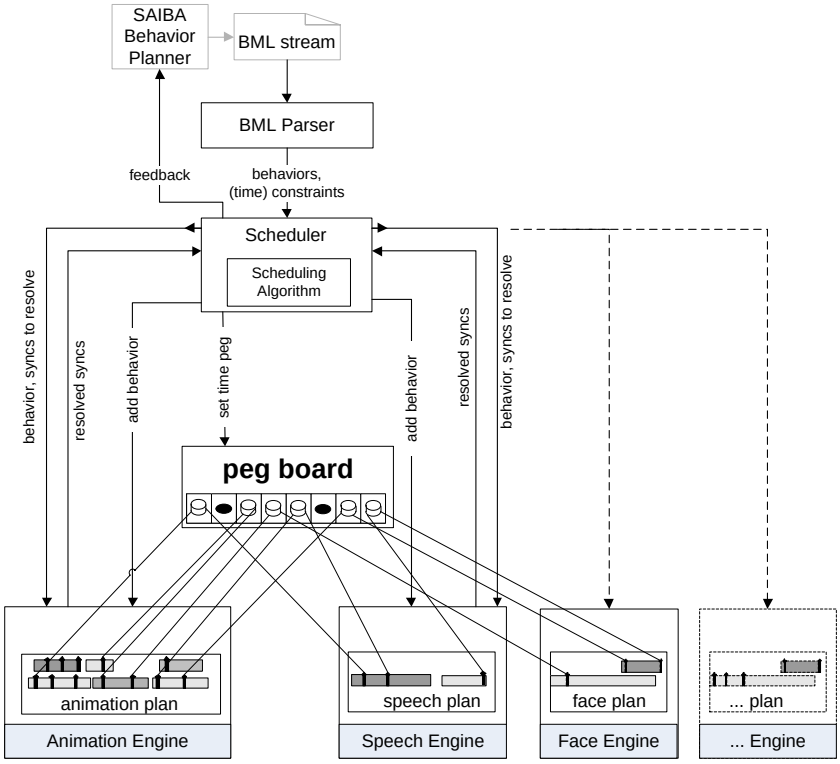


Fig. 4: A simplified view of Elckerlyc's scheduling architecture.

Interfacing with the Engines Elckerlyc's scheduler communicates with the Engines through their abstract interface (see below). It knows for each BML behavior type which Engine handles it. The various Engines provide the central scheduler with detailed information on the possible timing of behaviors in their specific modality, given the BML description and time constraints. To this end, each Engine implements functionality to:

1. Add a BML behavior to its unimodal motor plan.

2. Resolve unknown time constraints on a behavior, given certain known time constraints.
3. Check which behaviors in the Plan (if any) are currently invalid (due to recent modifications elsewhere in the motor plan).
4. Modify parameters on Timed Play Units, given the id of the BML behavior that these units express (and the new parameter values)
5. Remove Timed Play Units, given the id of the BML behavior that these units express

An Engine can be queried for time constraints on a behavior without adding it to the plan. This allows a scheduler to try out multiple constraint configurations on each behavior before it commits to a specific motor plan.[3] All communication with the Engine is in terms of BML behaviors. It is up to the Engine to map the BML behaviors to Timed Plan Units. The validity check is typically used to check if a motor plan is still valid after the timing of behaviors has been modified, in this modality or in another modality.

Scheduling Algorithm Elckerlyc was designed to be configurable with regards of the actual scheduling mechanism that it uses: the BML parsing and block management are separated from the scheduling algorithm, and the Engines provide generic interfaces that provide a scheduling algorithm with the timing of unknown constraints on behaviors, given certain known constraints.

The scheduler delegates the actual scheduling to a dedicated algorithm class that assigns (a prediction of) the timing of all Timed Plan Units that result from adding a new BML block, given the current multimodal motor plan and a parsed BML block that is to be scheduled. Elckerlyc currently uses an improved version of SmartBody's scheduling algorithm to do this, in which the behaviors in a new BML block are first sorted with respect to the flexibility of the behavior type, to avoid the ordering problem discussed in Section 4. However, this scheduling algorithm can easily be replaced by other algorithms (e.g., a custom constraint solver such as that of EMBR).

5.2 Elckerlyc's Plan Representation

Central to Elckerlyc's plan representation is the Peg Board shown in Fig. 4. Here we describe the relations between the elements on the Peg board; a graphical representation of the relations is shown in Fig. 5.

The sync points of each Timed Plan Unit in the motor plan are associated with Time Pegs on the Peg Board. These Time Pegs can be moved, automatically changing the timing of the associated sync points. If two sync points are connected by an 'at' constraint, they share the same Time Peg. This Time Peg can then be moved without violating the 'at' constraint, because this simultaneously changes the actual time of both sync points.

[3] Our current SmartBody-based scheduling strategy does not make use of this functionality yet, it is provided for future extension.

Each BML Block has its own associated BML Block Peg that defines the global start time of that block. Time Pegs are linked to their associated BML Block Peg, and thus provide local timing (that is, as offset from the start of the block). If the BML Block Peg is moved, all Time Pegs associated with it move along. This allows one to move the block as a whole, implicitly keeping the intra-block constraints consistent (see Fig. 5). The actual time of a BML Block Peg is first estimated to be *ct* (the time at which it is being scheduled). When playback of the Block Peg is started, its time is updated to reflect its actual start time. Since the Time Pegs inside the block are attached to the Block Peg, they will now also adhere to the actual start time of the block.

Some behaviors have constraints (and thus Time Pegs) that are linked to external global Pegs, used to synchronize behavior with external events. These are hooked up to a special, unmovable global BML Block Peg at global $t = 0$.

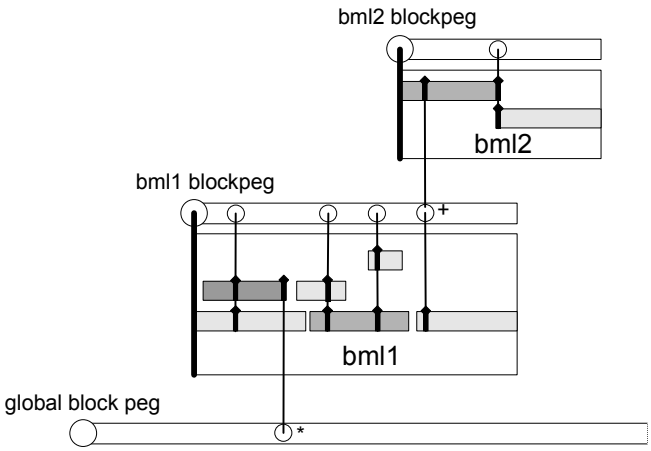


Fig. 5: Each BML block has its associated BML Block Peg. Internal constraints are linked to Time Pegs associated with this BML Block Peg. BML block `bml1` contains a constraint that is linked to an external Time Peg (marked with *). BML block `bml2` is block scheduled with the tight-merge scheduling algorithm. It has a constraint whose timing is defined by a Time Peg from BML block `bml1` (marked with +).

5.3 Resolving Constraints to Time Pegs

In Elckerlyc, scheduling consists of resolving the constraints in a BML block to Time Pegs, and assigning the Time Pegs a first prediction of their execution time. Relative 'at' synchronization constraints that share a sync point (behavior id, sync id pair) should be connected to the same Time Peg. Such 'at' constraints may involve a fixed, nonzero timing offset, for example when a nod is constrained to occur exactly half a second after the stroke of a gesture. Such offsets are maintained by special "Offset Pegs". An Offset Peg is a Time

Peg that is restrained to stay at a fixed offset to its linked Time Peg. If the Offset Peg is moved, its linked Time Peg moves with it and vice-versa. Offset Pegs can also be added by the scheduler for other reasons. For example, if the `start` sync is not constrained in a behavior, it may be resolved as an Offset Peg. That is: the `start` sync of the Timed Plan Unit is linked to the closest Time Peg of another sync point within the same Timed Plan Unit. If this other Time Peg is moved, the `start` of the Timed Plan Unit is moved with it. If a behavior is completely unconstrained, a new Time Peg is created and connected to the `start` sync of its Timed Plan Unit. BML Example 3 shows how Time Pegs are resolved for an example BML constraint specification.

5.4 Managing Adjustments of the Behavior Plan during Behavior Playback

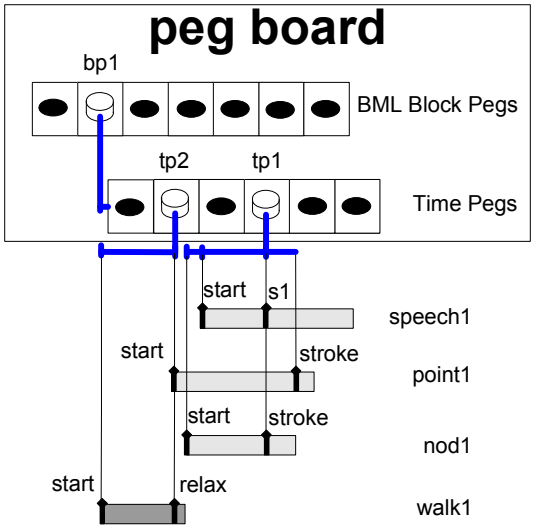
Once a BML block is scheduled, several changes can occur to its timing at playback time. Such changes may, for example, be initiated by a Time Peg being moved for external reasons (e.g., to postpone a speech phrase until the interlocutor finished uttering an assessment feedback, as explained in the introduction), or by other behaviors in the plan being removed. Since the sync points of behaviors are symbolically linked to the Time Pegs, timing updates are handled automatically (stretching or shortening the duration of behaviors when required) and the explicit constraints of Section 3.1 remain satisfied.

A dedicated BML Block management state machine automatically updates the timing of the BML Block Pegs in reaction to behavior plan modifications that occur at runtime, to maintain the BML Block constraints. For example, when a block b_i was scheduled to occur immediately after all behaviors already present in the motor plan, and the immediately preceding behaviors in the plan are removed from the plan through an `interrupt` behavior, the state machine will automatically move the BML Block Peg of b_i to close the resulting gap.

Plan changes, and constraint satisfaction after plan changes, are achieved in an efficient manner, that is, without requiring a time consuming scheduling action for minor plan adjustments. Interrupting a behavior in a BML block might shorten the length of the block. Since the BML Block management state machine dynamically manages the block end, shortening the block whenever this happens, the cluster constraint and append constraints automatically remain satisfied.

More significant updates might require re-scheduling of behaviors, such as when a Time Peg, linked to the start of a behavior, is moved to occur *after* the end of the same behavior. To check for such situations, the Scheduler asks each Engine whether its current plan is still valid (i.e., its constraints are still satisfied). The Scheduler then omits the behaviors that are no longer valid and notifies the SAIBA Behavior Planner using the BML feedback mechanism. It will then be up to the SAIBA Behavior Planner to update the behavior plan (using BML), if desired.

BML Example 3 Resolving a BML constraint specification to a Time Pegs specification. A Time Peg $tp1$ connects relative 'at' constraints $[[[bml1, speech1], s1], [[bml1, nod1], stroke], 0]$, and $[[[bml1, speech1], s1], [[bml1, point], stroke], -0.5]$. Another Time Peg $tp2$ is created for the 'at' constraint $[[[bml1, point1], start], [[bml1, walk1], relax], 0]$. Since the start time of `speech1`, `nod1`, and `walk1` is not constrained, they are attached to an Offset Peg linked to the closest other Time Peg in the respective behaviors. The BML Block itself (with id `bml1`) is connected to BML Block Peg `bp1`. All Time Pegs are connected to this Block Peg.




```
<bml id="bml1">
    <speech id="speech1">
      <text>As you can see, this <sync id="s1"> beautiful vase ...</text>
    </speech>
    <gesture id="point1" start="walk1:relax" type="POINT"
    target="vase1" stroke="speech1:s1+0.5"/>
    <head id="nod1" action="ROTATION" rotation="NOD" stroke="speech1:s1"/>
    <locomotion id="walk1" target="vase1"/>
</bml>
```

6 Employing the flexible plan representation

In the previous sections we described Elckerlyc's capabilities for on-the-fly adjustments to its multimodal behavior plans. We have been experimenting with these capabilities in a number of applications and proof of concept scenarios. The latest version of our Reactive Virtual Trainer performs fitness exercises along with the user, adjusting the timing of its performance to that of the user (Dehling 2011). In our experiments on Attentive Speaking, a route guide slightly delays its speech to make space for listener responses from the user

(Reidsma et al 2011) (using a Wizard of Oz setup for detecting start and end of listener responses). We have implemented a proof of concept setup for graceful interruption, in which the user can enter a text for the virtual human to speak, and then, while the virtual human is speaking, interrupt it – after which the virtual human finishes its current syllable, then completes the current word on a slightly lower pitch and volume than originally planned (parameter adaptation), and finally drops the remainder of the sentence. A number of other applications and scenarios have been described elsewhere; videos and demonstrations may be found on the Elckerlyc web site and in the open source code release. Our scheduling approach has recently been adapted in the Thalamus robotic framework developed at the Technical University of Lisbon (Ribeiro et al 2012), where it provides flexible (input) event based control of interactive robots.

In this section we explore a bit further the possibilities and consequences of the increased flexibility of our platform.

6.1 Continuous (re)scheduling

Rather than scheduling the plan only once, the plan could be rescheduled every execution step (or more efficiently at events that trigger a plan change). This would result in a constraint satisfaction that is more flexible to change than the one currently used in Elckerlyc. Furthermore, the resulting plan would always be the most natural one (for some measure of naturalness). However, continuous (re)scheduling is calculation time intensive and might not reflect actual human behavior. This approach also requires one to design a reward function for the naturalness of the motion plan. This is already challenging for one modality; designing one that models cross-modal naturalness in a non-adhoc fashion is probably infeasible. Nevertheless, is might be interesting to explore this approach further.

6.2 Combining top down scheduling with bottom up scheduling in Elckerlyc.

In Section 4.2 we described the flexible bottom up scheduling approach employed in the ACE system (Kopp and Wachsmuth 2004). In Figure 6 we illustrate how this scheduling strategy is implemented using Elckerlyc's plan representation and a combination of one top-down scheduling step to construct a flexible motor plan, another top-down scheduling step to make last-minute adjustments to the ongoing plan, and continuous bottom-up plan updates to modify the timing of unconstrained elements (e.g. the timing of the preparation and retraction duration of gestures). This scheduling approach has been implemented in Elckerlyc's successor AsapRealizer; we refer the interested reader to (van Welbergen et al 2012) for implementation details. Our current implementation only shifts the start time of speech behaviors, their inner timing is left unmodified.

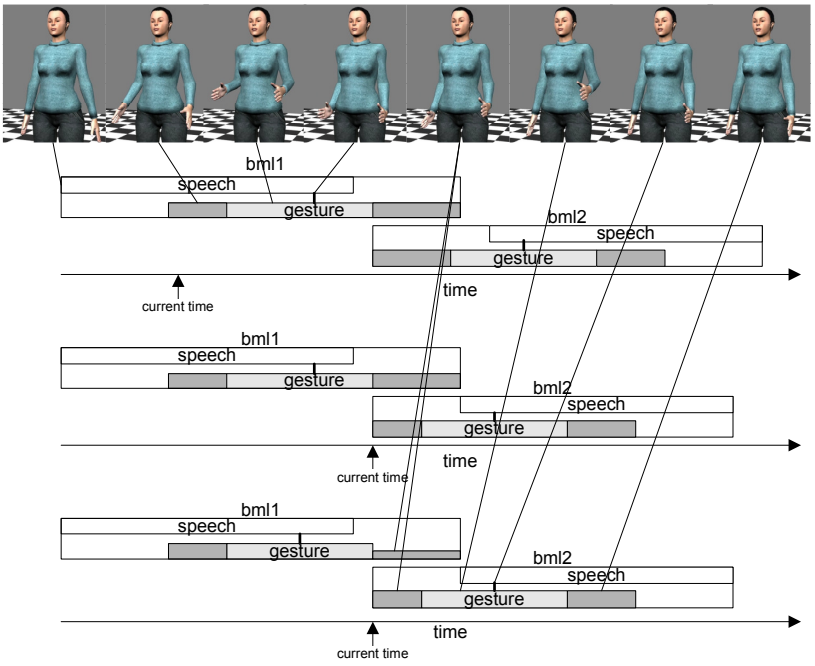


Fig. 6: An example of gesture chunking in AsapRealizer: First BML block `bml1` is being executed and a preliminary plan for `bml2` is being created using a top down scheduling step (top plan graph). As `bml1` is subsiding, `bml2` is (in another top-down scheduling step) re-aligned to fit the current behavior state (middle). This involves shortening the gesture preparation since the hand is still in gesture space. As the gesture of `bml1` is being retracted, it has a lower priority than the preparation of the gesture of `bml2` and is overridden by it (bottom plan graph). Since `bml2`'s gesture acts only on the left hand, a cleanup motion is generated for the right hand part of `bml1`'s gesture. Continuous bottom-up plan updates are used to adapt the preparation and retraction times of the gestures to the current hand position and rest state respectively.

However, the plan represention proposed in this work can provide a further generalization to this strategy: it is not limited to synchronizing one modality with flexible timing to one with completely fixed timing: instead it can provide synchronization strategies between multiple modalities that all provide flexible re-timing. In Figure 7 illustrates a scenario in which speech generated by a speech synthesis system that allow speech re-timing (e.g. (Baumann and Schlangen 2012)) to a flexible gesture. When the synchronization of two or more flexible modalities is managed in a flexible manner, multiple engines might be updating the same TimePeg. A conflict management strategy should be designed to handle such conflicting update requests. It could, for example, give precedence to updates from less flexible engines, or set a weighted average of the values set by different engines. The management strategy (or the PegBoard itself) should at the very least keep track of which process (e.g. top down scheduling, engine X) has set the value of a certain TimePeg. This allows engines to identify and overwrite their previous predictions of TimePeg

values. Furthermore, the management strategy could keep track of /use certain features of engines and/or behaviors that can be used to calculate a joint time prediction for synchronization points shared by multiple behaviors. What exactly these features are, and how to design good conflict resolution strategies will be determined in future work. A first simple strategy – that captures ACE's functionality – would be to encode the flexibility of each engine and give the least flexible engine precedence when updating shared TimePeg values.

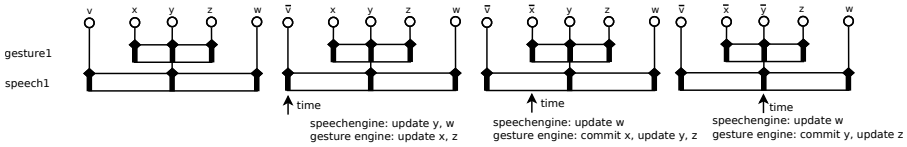


Fig. 7: Synchronizing a flexible speech behavior with a gesture. In the first scheduling step (left), the plan representation is constructed. TimePeg x, y and z are connected to the start, stroke and end of gesture1 respectively. The affiliate of speech1 is connected to the stroke of gesture1. Since the timing of speech is completely flexible, speech1 has its start and end synchronization points connected to two seperate TimePegs v and w. In the second scheduling step, it is determined that the bml block starts with speech1. This commits the timing of TimePeg v. As the bml block is being executed, the timing of its uncommited TimePegs is continuously updated (3 rightmost steps). Note that TimePeg y can potentially be updated by both the gesture engine and the speech engine. In this example, a simple conflict resolution mechanism is used: 1) the gesture engine will not update y as long as gesture1 is not running and x can be achieved given an y provided by the speech engine; 2) the speech engine will not update TimePegs that have been assigned a preditive value by a less flexible engine (here the gesture engine). This results in the speech engine updating y as long as gesture1 is not started, and the gesture engine updating y thereafter.

6.3 Multithreaded Scheduling

Currently, Elckerlyc schedules one BML block at a time, and the blocks are scheduled in order of arrival. The blocks that are to be scheduled form a scheduling queue. If a new BML block is appended to the scheduling queue, it will not be scheduled until the scheduling of the other blocks is finished. This can potentially ruin the rapid interruptibility and adaptability we strive for. If, for example, the agent is currently speaking, and we have just sent off a few more long sentences that we need to be uttered next, the scheduler may be occupied for a while. If then the virtual human's interlocutor smiles, and we *immediately* want the virtual human to smile back, we send a BML block containing a smile behavior to the realizer – but the scheduler is occupied and cannot add the smile to the plan in time! Obviously, in that situation, the scheduling of the new BML block with the smile should not have been delayed until the scheduling of the (unrelated) other blocks of speech are finished.

However, it is not actually necessary for the realizer to schedule the BML blocks in order. Scheduling of any new BML block can be started as soon as there are no *dependent* BML blocks in front of it in the scheduling queue.

A BML blockbmlY is dependent on BML block bmlX if:

1. bmlX is in front of bmlY in the queue, and
2. bmlX and bmlY share any constraints

From Section 3 and Appendix A it may be seen that bml_x and bml_y share any constraints if:

1. bmlY is appended after bmlX, or
2. bmlY interrupts bmlX, or
3. one or more time constraints in bmlY refer to bmlX (directly or indirectly), or
4. one or more behaviors in bmlY refer to bmlX (currently only for interrupt behaviors and parameter change behaviors)

All these situations can be found by parsing (rather than scheduling) the BML block.

Elckerlyc's scheduler can be extended to a multi-threaded scheduler that spawns new scheduling threads for all independent BML blocks in the queue. Whenever a new BML block is added to the queue, or scheduling of a BML block is finished, the scheduler will check the current queue of BML blocks and their parsed constraints and spawn a scheduling thread for all BML blocks that have no more dependencies on other unscheduled blocks.

7 Conclusion

We showed in this paper how the BML scheduling process can be viewed as a constraint problem, and how Elckerlyc uses this view to maintain a flexible behavior plan representation that allows one to make on-the-fly adjustments to behaviors while maintaining adherence to constraints. In Elckerlyc, scheduling is modeled as an interplay between different unimodal Engines that provide detailed information on the timing of the behaviors that are to be realized. The seperation of concerns between unimodal behavior timing, BML parsing, BML block progress management and multimodal scheduling makes it easy to exchange Elckerlyc's scheduling algorithm by a different one as well as to add new modalities. Thanks to the capability for on-the-fly plan adjustments, Elckerlyc is eminently suitable for Virtual Human applications in which a tight mutual coordination between user and Virtual Human is required.

References

Baumann T, Schlangen D (2012) Inpro_iss: A component for just-in-time incremental speech synthesis. In: Proceedings of the ACL System Demonstrations, Association for Computational Linguistics, pp 103–108

Dehling E (2011) The reactive virtual trainer. Master's thesis, University of Twente, Enschede, the Netherlands

Goodwin C (1986) Between and within: Alternative sequential treatments of continuers and assessments. Human Studies 9(2-3):205–217, DOI 10.1007/bf00148127

Heloir A, Kipp M (2010) Real-time animation of interactive agents: Specification and realization. Applied Artificial Intelligence 24(6):510–529, DOI 10.1080/08839514.2010.492161

Kipp M, Heloir A, Schröder M, Gebhard P (2010) Realizing multimodal behavior: Closing the gap between behavior planning and embodied agent presentation. In: Intelligent Virtual Agents, Springer, LNCS, vol 6356, pp 57– 63

Kopp S (2010) Social resonance and embodied coordination in face-to-face conversation with artificial interlocutors. Speech Communication 52(6):587 – 597, DOI doi:10.1016/j.specom.2010.02.007

Kopp S, Wachsmuth I (2004) Synthesizing multimodal utterances for conversational agents. Computer Animation and Virtual Worlds 15(1):39–52, DOI 10.1002/cav.v15:1

Kopp S, Krenn B, Marsella SC, Marshall AN, Pelachaud C, Pirker H, Thórisson KR, Vilhjálmsson HH (2006) Towards a common framework for multimodal generation: The behavior markup language. In: Intelligent Virtual Agents, Springer, LNCS, vol 4133, pp 205– 217

Nijholt A, Reidsma D, van Welbergen H, op den Akker H, Ruttkay ZM (2008) Mutually coordinated anticipatory multimodal interaction. In: Verbal and Nonverbal Features of Human-Human and Human-Machine Interaction, Springer Verlag, Berlin, pp 70–89

Reidsma D, de Kok I, Neiberg D, Pammi S, van Straalen B, Truong K, van Welbergen H (2011) Continuous interaction with a virtual human. Journal on Multimodal User Interfaces 4:97–118, DOI 10.1007/s12193-011-0060-x

Ribeiro T, Vala M, Paiva A (2012) Thalamus: Closing the mind-body loop in interactive embodied characters. In: Intelligent Virtual Agents, Springer, LNCS, vol 7502, pp 189–195

Thiebaux M, Marshall AN, Marsella SC, Kallmann M (2008) Smartbody: Behavior realization for embodied conversational agents. In: Proc. AAMAS, pp 151–158

van Welbergen H, Reidsma D, Ruttkay ZM, Zwiers J (2010) Elckerlyc: A BML realizer for continuous, multimodal interaction with a virtual human. Journal on Multimodal User Interfaces 3(4):271–284, DOI 10.1007/s12193-010-0051-3

van Welbergen H, Xu Y, Thiébaux M, Feng WW, Fu J, Reidsma D, Shapiro A (2011) Demonstrating and testing the bml compliance of bml realizers. In: Intelligent Virtual Agents, Springer, LNCS, vol 6895, pp 269–281, DOI 10.1007/978-3-642-23974-8_30

van Welbergen H, Reidsma D, Kopp S (2012) An incremental multimodal realizer for behavior co-articulation and coordination. In: Intelligent Virtual Agents, LNCS, vol 7502, pp 175–188, DOI 10.1007/978-3-642-33197-8_18

Zwiers J, van Welbergen H, Reidsma D (2011) Continuous interaction within the SAIBA framework. In: Intelligent Virtual Agents, Springer, Lecture Notes in Computer Science, vol 6895, pp 324–330, DOI 10.1007/978-3-642-23974-8_35

A Full constraint descriptions

A.1 Explicit Constraints

A *sync ref* consists of either an offset from the start of the BML block, or a pair $[s, o]$, where s is a sync point, defined by the pair $[b, \text{sync id}]$ and o is a time offset (in seconds) from the time of the sync id. b is defined as [block id, behavior id].

For ease of specification and without loss of generality, we define each time constraint as acting between two sync refs. A constraint is an absolute constraint if one of the sync refs is an offset from the start of the BML block. A constraint is a relative constraint if both sync refs are triples of behavior id, sync id and offset time.

An absolute 'at' constraint c_a on a sync point with id s in behavior b at offset o from the start of the BML block is defined by

$$c_a = [[b, s], o] \tag{2}$$

Absolute before and after constraints c_{a_b} and c_{a_a} on a sync point with id s in behavior b at offset o from the start of the BML block are defined as

$$c_{a_b} = [[b, s], o] \tag{3}$$

$$c_{a_a} = [[b, s], o] \tag{4}$$

A relative 'at' constraint c_r between sync refs $[[b_1, s_1], o_1]$ and $[[b_2, s_2], o_2]$ is defined by

$$c_r = [[b_1, s_1], [b_2, s_2], o_2 - o_1] \tag{5}$$

Relative before c_{r_b} and relative after c_{r_a} constraints between sync refs $[[b_1, s_1], o_1]$ and $[[b_2, s_2], o_2]$ are defined as follows:

$$c_{r_b} = [[b_1, s_1], [b_2, s_2], o_2 - o_1] \tag{6}$$

$$c_{r_a} = [[b_1, s_1], [b_2, s_2], o_2 - o_1] \tag{7}$$

A relative before constraint $[[b_1, s_1], [b_2, s_2], o]$ can be converted to relative after constraint c_{r_a} using

$$c_{r_a} = [[b_2, s_2], [b_1, s_1], -o] \tag{8}$$

A BML block contains a set of behaviors $\mathbf{b}$, a set of sync points (pairs of behavior id and sync id) $\mathbf{s}$, a set of absolute constraints $\mathbf{c_a}$, a set of absolute before constraints $\mathbf{c_{a_b}}$, a set of absolute after constraints $\mathbf{c_{a_a}}$, a set of relative constraints $\mathbf{c_r}$ and a set of relative after constraints $\mathbf{c_{r_a}}$.[4]

The function $f : \mathbf{s} \rightarrow \mathbf{t}$ maps a sync point s to global time t. The goal of scheduling is to find such a mapping for all sync points in all behaviors in the block in such a way that all constraints are satisfied. The function blockstart : $\mathbf{u} \rightarrow \mathbf{t}$ maps the block id u to its global start time t. In Section 3.5 we show how the blockstart is defined, given the composition attribute of the BML block.
The BML block defines the following explicit constraints on f:

$$\forall[[[\text{bmlid}, \text{behid}], s], o] \in \mathbf{c_a}.\, f([\text{bmlid}, \text{behid}], s) = o + \text{blockstart}(\text{bmlid}) \tag{9}$$

$$\forall[[[\text{bmlid}, \text{behid}], s], o] \in \mathbf{c_{a_a}}.\, f([\text{bmlid}, \text{behid}], s) \geq o + \text{blockstart}(\text{bmlid}) \tag{10}$$

$$\forall[[[\text{bmlid}, \text{behid}], s], o] \in \mathbf{c_{a_b}}.\, f([\text{bmlid}, \text{behid}], s) \leq o + \text{blockstart}(\text{bmlid}) \tag{11}$$

$$\forall[[b_1, s_1], [b_2, s_2], o] \in \mathbf{c_r}.\, f(b_1, s_1) + o = f(b_2, s_2) \tag{12}$$

$$\forall[[b_1, s_1], [b_2, s_2], o] \in \mathbf{c_{r_a}}.\, f(b_1, s_1) + o \geq f(b_2, s_2) \tag{13}$$

A.2 Implicit Constraints

Besides the explicit constraints defined in the BML block, several implicit constraints act upon f:

1. Sync points may not occur before the block they are in is started (equation 14).
2. Behaviors should have a nonzero duration (equation 15).
3. The default BML sync points of each behavior must stay in order (equation 16).

[4] To specify the explicit constraints in a BML block in a unique manner, all relative before constraints are converted to after constraints using equation 8.

$$\forall[[\text{bmlid}, \text{behid}], s] \in \mathbf{s}.\, f([\text{bmlid}, \text{behid}], s]) \geq \text{blockstart}(\text{bmlid}) \tag{14}$$

$$\forall b \in \mathbf{b}.\, f([b, end]) > f([b, start]) \tag{15}$$

$$\begin{aligned} \forall b \in \mathbf{b}. & f([b, start]) \geq f([b, ready) \geq \\ & f([b, strokestart]) \geq f([b, stroke]) \geq \\ & f([b, strokeend]) \geq f([b, relax]) \geq f([b, end]) \end{aligned} \tag{16}$$

A.3 Cluster Constraints

A BML block may contain several clusters of behaviors. Each cluster contains a set of behavior connected with 'at' constraints. We define the start of the cluster as the start of the first behavior in the cluster. A cluster can be *grounded*[5], that is, connected to the start of a BML block with an absolute 'at' constraint, or ungrounded.

A scheduler has the freedom to set up the internal timing of each behavior as it likes, as long as the implicit and explicit constraints defined in the sections above are satisfied. This freedom is typically used to set up the timing of behaviors in such away that the resulting motor behavior is natural. One would like to schedule ungrounded clusters in such a way that gaps between clusters, or, between clusters and the start of the block are minimized, so that they start 'as soon as possible', while retaining this scheduling freedom.

The cluster constraint achieves this by setting up the constraint as one that acts between clusters, without requiring changes to the relative timing of the behavior *within* a cluster.

A.3.1 Cluster Properties

To define the cluster constraint formally, we introduce some predicates that indicate the cluster properties of a behavior.

The predicate $\text{DirectLink}(b_1, b_2)$ expresses that two behaviors b_1 and b_2 are directly connected by an 'at' constraint.

$$\begin{aligned} \text{DirectLink}(b_1, b_2) \equiv \exists o, s_1, s_2.\, ([[b_1, s_1], [b_2, s_2], o] \in \mathbf{c_r} \vee \\ [[b_2, s_2], [b_1, s_1], o] \in \mathbf{c_r}) \end{aligned} \tag{17}$$

The predicate $\text{IsConnected}(c, d)$ expresses that two behaviors c and d are connected by a chain of 'at' constraints.

$$\begin{aligned} \text{IsConnected}(c, d) \equiv \\ \exists N > 0.\, \forall i \in 0..N-1.\, b_i \in \mathbf{b} \wedge \text{DirectLink}(b_i, b_{i+i}) \wedge c = b_0 \wedge d = b_N \end{aligned} \tag{18}$$

The predicate DirectGround expresses that a behavior b has an absolute constraint.

$$\text{DirectGround}(b) \equiv \exists o, s.\, [[b, s], o] \in \mathbf{c_a} \tag{19}$$

The predicate DirectAfterGround expresses that a behavior b has an absolute after constraint.

$$\text{DirectAfterGround}(b) \equiv \exists o, s.\, [[b, s], o] \in \mathbf{c_{a_a}} \tag{20}$$

The predicate IsGrounded(b) expresses that a behavior b is part of a grounded cluster of behaviors.

$$\begin{aligned} \text{IsGrounded}(b) \equiv \\ \text{DirectGround}(b) \vee \exists c.\, (\text{IsConnected}(b, c) \wedge \text{DirectGround}(c)) \end{aligned} \tag{21}$$

[5] The notion of grounding was taken from (Kipp et al 2010).

The predicate OnBlockStart([bmlid, behid]) expresses that the cluster of behavior [bmlid, behid] starts at blockstart(bml1).

$$\begin{aligned} &\text{OnBlockStart}([\text{bmlid}, \text{behid}]) \equiv \\ &f([[\text{bmlid}, \text{behid}], \mathit{start}]) = \text{blockstart}(\text{bmlid}) \vee \\ &(\exists c \in \mathbf{b}.\, \text{IsConnected}([\text{bmlid}, \text{behid}], c) \wedge f([c, \mathit{start}]) = \text{blockstart}(\text{bmlid})) \end{aligned} \tag{22}$$

The predicate OnAbsAfterConstraint(b) expresses that the cluster of behavior b satisfies one of its absolute after constraints as an at constraint.

$$\begin{aligned} &\text{OnAbsAfterConstraint}([\text{bmlid}, \text{behid}]) \equiv \\ &\exists [[b_1, s_1], o] \in \mathbf{c_{a_a}}.\, (([\text{bmlid}, \text{behid}] = b_1 \vee \\ &\qquad \text{IsConnected}([\text{bmlid}, \text{behid}], b_1)) \wedge \\ &\qquad f([b_1, s_1]) + o = \text{blockstart}(\text{bmlid})) \end{aligned} \tag{23}$$

The predicate OnRelAfterConstraint(b) expresses that the cluster of behavior b satisfies one of its relative after constraints as an at constraint.

$$\begin{aligned} &\text{OnRelAfterConstraint}([\text{bmlid}, \text{behid}]) \equiv \\ &\exists [[b_1, s_1], [b_2, s_2], o] \in \mathbf{c_{r_a}}.\, (([\text{bmlid}, \text{behid}] = b_1 \vee \\ &\qquad \text{IsConnected}([\text{bmlid}, \text{behid}], b_1)) \wedge \\ &\qquad f(b_1, s_1) = f(b_2, s_2) + o) \end{aligned} \tag{24}$$

A.3.2 The Cluster Constraint

An ungrounded cluster may contain relative or absolute 'after' constraints. If the gaps between clusters are to be minimized using only one constraint per cluster, this means that the cluster should start at the start of the BML block it is in, or that one of its 'after' constraints is satisfied as an 'at' constraint. If an ungrounded cluster has no 'after' constraints, then it should start at the start of the BML block it is in.

Using the cluster properties defined above, this cluster constraint is defined as:

$$\begin{aligned} &\neg\, \text{IsGrounded}([\text{bmlid}, \text{behid}]) \rightarrow \\ &\quad \text{OnBlockStart}([\text{bmlid}, \text{behid}]) \vee \\ &\quad \text{OnAbsAfterConstraint}([\text{bmlid}, \text{behid}]) \vee \\ &\quad \text{OnRelAfterConstraint}([\text{bmlid}, \text{behid}]) \end{aligned} \tag{25}$$

A.4 Block Level Constraints

The composition attribute defined in the BML Block defines constraints on the start of the block in relation to the set of current behaviors in the multimodal behavior plan **B** and the current global time ct. Core BML defines the following scheduling attributes:

1. `merge`: start the block at ct (equation 26).
2. `replace`: completely replaces the current behavior, start the block at ct (equation 27).
3. `append`: start the block directly after all behaviors in the current plan are finished (equation 28).

$$\text{compositionattribute}(\text{bml1}) = \text{merge} \rightarrow \text{blockstart}(\text{bml1}) = \text{ct} \tag{26}$$

$$\text{compositionattribute}(\text{bml1}) = \text{replace} \rightarrow \text{blockstart}(\text{bml1}) = \text{ct} \tag{27}$$

$$\begin{aligned} &\text{compositionattribute}(\text{bml1}) = \text{append} \rightarrow \text{blockstart}(\text{bml1}) \geq \text{ct} \wedge \\ &\qquad \forall b \in \mathbf{B}.\, f(b, \mathit{end}) \leq \text{blockstart}(\text{bml1}) \wedge \\ &((\exists b \in \mathbf{B}.\, f(b, \mathit{end}) = \text{blockstart}(\text{bml1})) \vee (\text{blockstart}(\text{bml1}) = \text{ct})) \end{aligned} \tag{28}$$

A.5 Additional Behavior Plan Constraints In Elckerlyc

Elckerlyc provides an extension to BML, BML$^{\mathrm{T}}$, that, among other things, allows the specification of additional behavior constraints.

A.5.1 Anticipator Constraints

Elckerlyc's multimodal behavior plan is designed to allow micro adjustments in its timing. Such time adjustments are often steered by Anticipators. An Anticipator instantiates synchronization points that can be used in BML blocks to constrain the timing of behaviors. It uses perceptions of events in the real world to continuously update the timing of its sync points, by extrapolating the perceptions into *predictions* of the timing of future events. An anticipator sync a is defined by $a = [\text{anticipatorid}, \text{syncid}]$.

Constraint c_{ant} describes an 'at' constraint on sync with id s in behavior b at offset o from the anticipator sync a.

$$c_{ant} = [[b, s], o, a] \tag{29}$$

A sync point should be connected to at most one anticipator sync with an 'at' constraint.

Constraint c_{ant_a} describes an 'after' constraint on sync with id s in behavior b at offset o from the anticipator sync a.

$$c_{ant_a} = [[b, s], o, a] \tag{30}$$

Constraint c_{ant_b} describes a 'before' constraint on sync with id s in behavior b at offset o from the anticipator sync a.

$$c_{ant_b} = [[b, s], o, a] \tag{31}$$

In addition to a set of behaviors $\mathbf{b}$, a set of sync points (pairs of behavior id and sync id) $\mathbf{s}$, a set of absolute constraints $\mathbf{c_a}$, a set of absolute before constraints $\mathbf{c_{a_b}}$, a set of absolute after constraints $\mathbf{c_{a_a}}$, a set of relative constraints $\mathbf{c_r}$ and a set of relative after constraints $\mathbf{c_{r_a}}$, a BML$^{\mathrm{T}}$ block contains a set of anticipator syncs $\mathbf{a}$, a set of Anticipator constraints $\mathbf{c_{ant}}$, a set of Anticipator after constraints $\mathbf{c_{ant_a}}$ and a set of Anticipator before constraints $\mathbf{c_{ant_b}}$.

Anticipators provide a global time for their sync points. The function $g : \mathbf{a} \rightarrow \mathbf{t}$ maps an Anticipator sync a to its global time t. The value of $g(a)$ is completely defined by the time prediction of a's Anticipator. Anticipator constraints add the following explicit constraint to the behavior plan:

$$\forall [[b, s], o, a] \in \mathbf{c_{ant}} . f([b, s]) + o = g(a) \tag{32}$$

$$\forall [[b, s], o, a] \in \mathbf{c_{ant_a}} . f([b, s]) + o \geq g(a) \tag{33}$$

$$\forall [[b, s], o, a] \in \mathbf{c_{ant_b}} . f([b, s]) + o \leq g(a) \tag{34}$$

A.5.2 Cluster Constraints

Anticipators extend Elckerlyc's notion of 'grounding'. In Elckerlyc, a behavior is grounded not only if it is connected to an absolute 'at' constraint but also if it is connected to an Anticipator sync point. The DirectGround predicate is updated to reflect this (see equation 35).

$$\text{DirectGround}(b) \equiv \exists o, s. [[b, s], o] \in \mathbf{c_a} \vee \exists o, s. [[b, s], o, a] \in \mathbf{c_{ant}} \tag{35}$$

The predicate OnAbsAfterAntConstraint(b) expresses that the cluster containing behavior b satisfies one of its absolute anticipator 'after' constraints as an 'at' constraint.

$$\begin{aligned} &\text{OnAbsAfterAntConstraint}([\text{bmlid}, \text{behid}]) \equiv \\ &\exists [[b_1, s_1], o, a] \in \mathbf{c_{a_{ant_a}}} . ((([\text{bmlid}, \text{behid}] = b_1 \vee \\ &\qquad \text{IsConnected}([\text{bmlid}, \text{behid}], b_1)) \wedge \\ &\qquad f([b_1, s_1]) + o = g(a)) \end{aligned} \tag{36}$$

The updated cluster constraint then becomes:

$$\begin{aligned} \neg\,\text{IsGrounded}([\text{bmlid}, \text{behid}]) \rightarrow \\ \text{OnBlockStart}([\text{bmlid}, \text{behid}]) \vee \\ \text{OnAbsAfterConstraint}([\text{bmlid}, \text{behid}]) \vee \\ \text{OnRelAfterConstraint}([\text{bmlid}, \text{behid}]) \vee \\ \text{OnAbsAfterAntConstraint}([\text{bmlid}, \text{behid}]) \end{aligned} \tag{37}$$

A.5.3 Block Level Constraint

In addition to the Core BML `merge` and `append` composition attributes, BML$^{\text{T}}$ provides the `append-after(X)` composition attribute. Append-after starts a BML block directly after a selected set of behaviors (those from a BML block in $\mathbf{X}$) in the current behavior plan are finished (equation 38).

$$\begin{aligned} \text{compositionattribute}(\text{bml1}) = \text{append-after}(\mathbf{X}) \rightarrow \\ \text{blockstart}(\text{bml1}) \geq \text{ct} \wedge \\ (\forall [\text{bmlid}, \text{behid}] \in \mathbf{B}.\, \text{bmlid} \in \mathbf{X} \rightarrow \\ f([\text{bmlid}, \text{behid}], \mathit{end}) \leq \text{blockstart}(\text{bml1})) \wedge \\ ((\exists [\text{bmlid}, \text{behid}] \in \mathbf{B}.\, f([\text{bmlid}, \text{behid}], \mathit{end}) = \text{blockstart}(\text{bml1}) \wedge \\ \text{bmlid} \in \mathbf{X}) \vee \\ (\text{blockstart}(\text{bml1}) = \text{ct})) \end{aligned} \tag{38}$$