Specifying Loops and Path Selection in Multimedia Presentations *

Robert Cooley, Joseph A. Konstan, Brian Bailey, Moses Dejong
{cooley,konstan,bailey,dejong}@cs.umn.edu
Department of Computer Science
University of Minnesota
4-192 EECS Bldg., 200 Union St. SE
Minneapolis, MN 55455, USA

February 22, 1998

Abstract

With the rapid technological advances in computer hardware and presentation devices that are occurring, the development of multimedia presentations is quickly expanding from the realm of professional studios to small offices and homes. Along with the hardware advances, much effort has been put into creating the necessary tools to convert raw data streams into polished presentations, including the specification of temporal relations between media objects (synchronization). Many models have been proposed and implemented for handling a rich set of both fine-grain (e.g. lip-synching an audio file with a video file) and coarse-grain (e.g. requiring object `A` to finish before object `B` starts) synchronization constraints. However, the explicit handling of loops and selection of one out of many paths within a synchronization model are important but often overlooked capabilities. This paper extends an existing coarse-grain synchronization model, the FLexible Interactive Presentation Synchronization (FLIPS) model [SKD96], to add these capabilities.

Keywords: multimedia, synchronization, coarse-grain

*This work was supported in part by NSF grants EHR-9554517 and IRI-9410470

1 Introduction and Background

With the rapid technological advances in computer hardware and presentation devices that are occurring, the development of multimedia presentations is quickly expanding from the realm of professional studios to small offices and homes. Affordable camcorders, digital cameras, and CPU's with dedicated multimedia instruction sets are among the products that have emerged within the last few years. Along with the hardware advances, much effort has been put into creating the software and synchronization tools to convert raw data streams into polished presentations for applications such as academic and industrial education or product information for consumers. Specification of a coherent, semantically meaningful presentation requires more than simple media players. For example, consider an on-line manual for a sport utility vehicle (SUV). The manual may consist of several sections dealing with subjects such as basic operation, 4-wheel drive operation, and climate control. Each section could contain a number of different media sources, such as audio, video, slides, and animation. Along with a method of delivering each type of media to the user, temporal relations between media objects must be addressed, such as fine-grain synchronization, which specifies close connections between two or more media streams (e.g. lip-synching an audio file with a video file). A popular method for handling such synchronization is to tie both media streams to a common clock [RS92]. Another type of synchronization, known as coarse-grain synchronization, deals with specific synchronization points but does not require media objects to be tightly coupled as with fine-grain synchronization. In between synchronization points each media object can proceed at a different pace. A coarse-grain synchronization model can be defined as part of, or indpendendantly of any fine-grain synchronization. In the context of the SUV manual scenario, some examples of coarse-grain synchronization would be:

- All of the basic use objects should be finished playing before the 4-wheel drive objects start (Sequential play).
- The 4-wheel drive slide should end when the 4-wheel drive video ends (Master-slave).
- The first climate control audio and animation objects should end at the same time (Parallel-last).

For media where the duration is known ahead of time, a timeline synchronization model can be used for the coarse-grain synchronization as well as the fine-grain. However, if the length of the media is not known ahead of time or is variable, timeline synchronization breaks down. Examples

of unknown or variable duration media are text or slides where the user chooses when to continue, media that must be played as it is down-loaded from an outside source, or simulations that depend on processor speed and load. Within a single media object, variable duration can be handled by slowing down or speeding up the clock, but this method may not make sense for more than one media object. For example, if a user quickly forwards through a sequence of slides, the background music should not skip or fast-forward along with the slides. The music should play at normal speed and simply stop when all of the slides have been viewed. Many models [CPS96, RH96, Dra93] can not handle variable duration media, and others [LG93, Paz96, BHL91, CO96] allow no or limited user interaction. One method for specifying coarse-grain synchronization, FLIPS (FLexible Interactive Presentation Synchronization) [SKD96], uses an event-based specification model and enforcement policy for handling variable duration media objects and user interactions. The FLIPS model is independent of any fine-grain synchronization necessary for the presentation. All of the examples of coarse-grain synchronization listed above can be handled by the FLIPS model.

Other types of coarse-grain synchronization constraints that a presentation designer should be able to specify are ones involving loops and path selection. Examples of these types of constraints are:

- The two background music objects should play in a loop until the entire presentation is finished (Loop).
- The user should be presented with a list of available options and be able to choose which one to view (Path selection).
- After a user has seen one segment, the presentation should loop back to the original selection point to allow the user to make addition choices (Selection-loop).
- The user should be able to view either the basic use or climate control segments immediately, but should not be able to view 4-wheel drive segment until the basic use segment has been seen (Restricted selection).

Restricted selection, as shown in the example above, refers to the ability to change the availability of certain paths in a presentation based on what the user has or has not seen at a given point. While these examples could be handled by embedding a synchronization model within a high-level programming language, it is useful for the designer to be able to completely specify a presentation

within a declarative synchronization model. Hypermedia systems, such as [BZ92, VJMB93], provide limited path selection capabilities, but generally do not include the concept of presentation history that is required to handle *restricted selection* specifications. The original FLIPS specification, which will be referred to as *basic* FLIPS in this paper, can not handle specifications like the ones listed above because it does not specifically address the concept of predefined loops within a presentation, or the handling of explicit path selection. This paper defines two extensions that increase the power of the *basic* FLIPS model while retaining the ability to handle user interaction and variable length media. The first, the *loop* construct, allows an n iteration loop to be specified within a presentation. The second, the *selection* construct, allows for one of many potential paths to be selected by the user for viewing. By combining a loop construct with a selection construct, a sequence of paths may be selected for playout by the user.

The rest of this paper is organized as follows: Section 2 briefly discusses the *basic* FLIPS model defined in [SKD96]. Section 3 presents extensions to the *basic* FLIPS model that allow the handling of loops and path selection. Finally, Section 4 provides conclusions.

2 Basic FLIPS

The *basic* FLIPS specification is made up media objects and two types of conditions called enablers and barriers. A media object is considered to contain a media file, such as an audio or video file. It is assumed that any required fine-grain synchronization is handled within the object, and is invisible to the coarse-grain synchronization model.

2.1 Enablers and Barriers

In order to specify temporal relations like the sequential play, master-slave, and parallel-last specifications shown in the introduction, a synchronization model must have a method of controlling and recognizing the begin and end of any media object. Control of the beginning of an object is done by specifying that a media object should begin playout as soon as possible after it is enabled and barrier-free. Enablers are essentially disjunctive or causation conditions, where if an object has `n` begin enablers, when any one or more of the enablers is activated, the object is considered begin-enabled. An enabler is usually activated by another object beginning or coming to an end. The situation where the end of object `A` will enable the begin of object `B` is shown in figure 1. Similar to the begin of an object, the end of an object can also be enabled. In the FLIPS specifi-

Figure 1: Sequential Play - the end of object `A` enables the beginning of object `B`

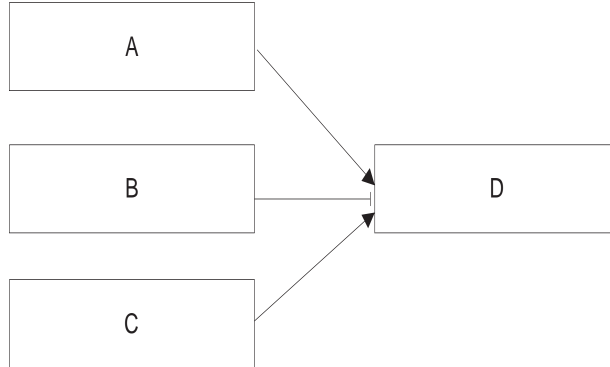

Figure 2: Enabler/Barrier Combination - `D` can be begin-enabled by either `A` or `C`. `D` is not begin barrier-free until `B` ends.

cation, an object that is end-enabled immediately stops, not waiting for the media file to finish its normal playout time. Every object is considered to have an implicit end enabler that activates at the end of the normal playout of a media file. Each begin or end enabler must have a source and a destination object. For the case of the implicit end enabler, the source and destination are the same object.

Because enablers are a disjunctive condition, they do not provide the capability to make specifications such as, object `C` can not start until both objects `A` and `B` have completed. For this reason FLIPS includes barriers, which are conjunctive or inhibiting conditions and prevent an action from occurring. The destination of a barrier can be specified as the begin or end of a media object, just like enablers. Activation of a barrier is referred to as removal, and an object that has all of its begin barriers activated is said to be begin barrier-free. As with enablers, an object can also have end barriers that prevent an object from being considered complete. If an object finishes its normal playout and is not end barrier-free, a default action can be defined such as holding the last frame of a video until the barriers are lifted. This concept is defined as *restrictive blocking* by Stienmetz [Ste90]. Combinations of begin and end barriers and enablers can provide very powerful synchronization constructs, including all seven types listed in [BS96]. Figure 2 shows a situation

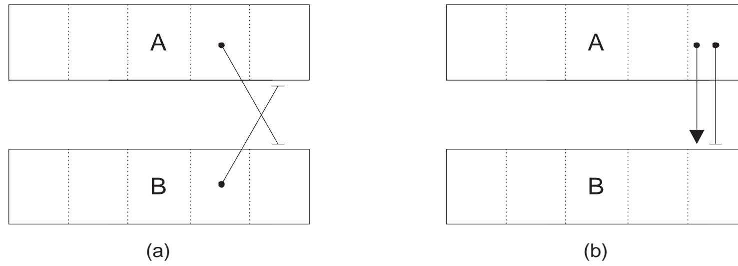

Figure 3: (a) Parallel-last - Neither A or B can complete until the other one finishes, (b) Master-slave - B must complete at the same time, but not before A.

where object D can not begin until either A or C have finished, and object B has finished. Note that while an object must have at least one enabler to start its playout, there do not have to be any barriers. An object with no begin barriers is automatically begin barrier-free. Also, when an object has multiple enablers, only the activation of the first enabler has any effect on the object. Subsequent enabler activations do not cause an object to restart. The current status of an object's begin and end barriers and enablers can be conveniently determined by defining five states as follows:

- An object is *idle* if it is not begin or end enabled.
- An object is *ready* if it is begin-enabled but not begin barrier-free.
- An object is *in-process* if it is begin-enabled and begin barrier-free, but not end-enabled.
- An object is *finished* if it is end-enabled, but not end barrier-free.
- An object is *complete* if it is end-enabled and end barrier-free.

During the normal playout of a presentation, enablers and barriers can only be activated, not revoked, meaning that an object's state can transition from *idle* toward *complete*, but never in the other direction. The source of any barrier or enabler can now be more specifically defined as coming from a specific state of an object. Figure 3 shows how the parallel-last and master-slave conditions can be specified with FLIPS. Figure 4 shows how the SUV online manual could be specified using FLIPS. An example of a media object, in this case the first climate control audio object, is shown in figure 5. An obvious problem with the state definitions is what happens if an end enabler is activated when an object is still in the *idle* state. How does it "finish" when it is still not allowed to start. To handle this case, FLIPS defines additional consistency rules stating that an object that is end-enabled must also be begin-enabled and begin barrier-free. Conversely, an object that

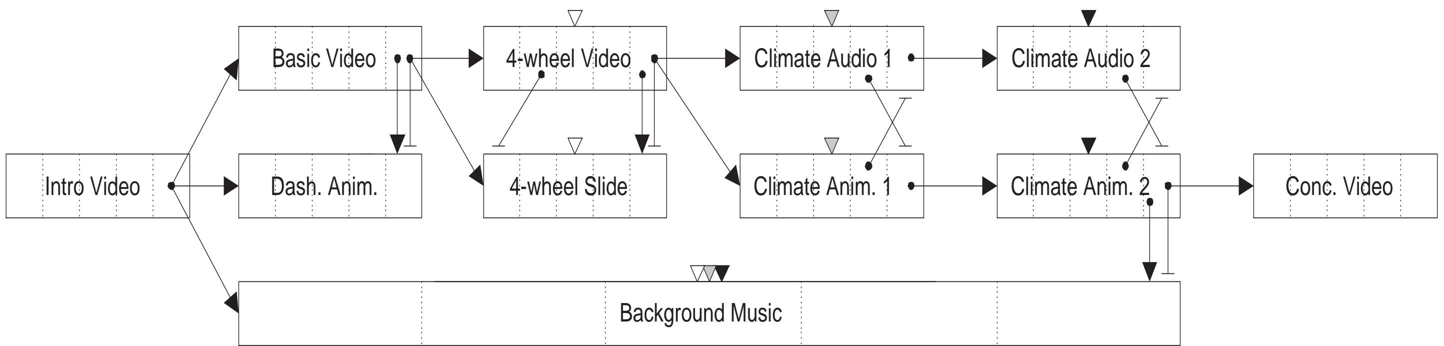

Figure 4: Sport Utility Vehicle On-line Manual - The end of the **introduction video** enables the beginning of the **basic video**, **dashboard animation**, and **background music**. The end of the **basic video** forces the **dashboard animation** to end and enables the beginning of both **4-wheel drive** objects. The end of the **4-wheel drive video** enables the beginning of the first set of **climate control** objects and forces the **4-wheel drive slide** to end. Both objects in each set of **climate control** objects must complete before the beginning of the next set is enabled. The end of the second **climate control animation** forces the **background music** to end and enables the beginning of the **conclusion video**. The white and black triangles show the in-process objects before and after a user skip forward from the **4-wheel drive** segment to the second **climate control** segment, respectively (See section 2.2). The grey triangles show the in-process objects after a user skip back to the first **climate control** segment.

Media Type:	*Pointer to Media:*	*Ideal Duaration:*	*State:*
audio	climate1.au	00:12:32	Idle

Media Specific Information:	*Alternative Action:*
none	none

Synchronization Information

Destination	**Source**	
Begin Enablers: 4w_video.complete	*Ready Enablers:* none	*Finish Enablers:* none
Begin Barriers: none	*Ready Barriers:* none	*Finish Barriers:* c_anim1.end
End Enablers: self	*In-process Enablers:* none	*Complete Enablers:* c_audio2.begin
End Barriers: c_anim1.finish	*In-process Barriers:* none	*Complete Barriers:* none

Figure 5: FLIPS Media Object - **climate audio 1** from figure 4 prior to the presentation reaching the first climate control segment.

is not begin-enabled or begin barrier-free can not be end-enabled. The method of enforcing these constraints is discussed in the next section.

2.2 Consistency

User interaction within an object is handled by whatever fine-grain synchronization model is being used. For instance, a user could fast-forward through a portion of a video, and besides changing the length of the normal playout, the coarse-grain synchronization is not affected. As soon as a user interaction moves the presentation past a synchronization point such as the source of an enabler or barrier, the synchronization model must ensure that the presentation remains globally consistent. For a skip forward, FLIPS handles this by propagating any necessary changes backward from the point the presentation has been moved to, until a consistent state is reached. An inconsistent state is one that is in violation of the two consistency rules listed in the previous section. Objects are forced into consistency by explicitly setting their state. As an example, if a user skips from the 4-wheel drive segment (marked with white triangles) to the second part of the climate control segment (marked with black triangles) in figure 4, FLIPS would take the following actions to restore global consistency:

- In order for **climate audio 2** to be *in-process*, **climate audio 1** must be *complete*.
- If **climate audio 1** is *complete*, then **climate animation 1** must also be *complete*.
- If **climate audio 1** and **climate animation 1** are both *complete*, then **4-wheel video** must be *complete*.
- If **4-wheel video** is *complete* then **4-wheel slide** must also be *complete*.
- If **climate animation 1** is *complete* then **climate animation 2** must be *in-process*.
- All objects are now in a consistent state.

This mechanism also explains how FLIPS handles the situation where an object is end-enabled while still in the *idle* or *ready* state. The necessary changes to the states of other objects are propagated backwards, as if a user skip occurred. The global consistency enforcement mechanism also works for user skips in the backwards direction. In this case, FLIPS starts from the point the presentation has been moved to and works forward until every object is in a consistent state. Since

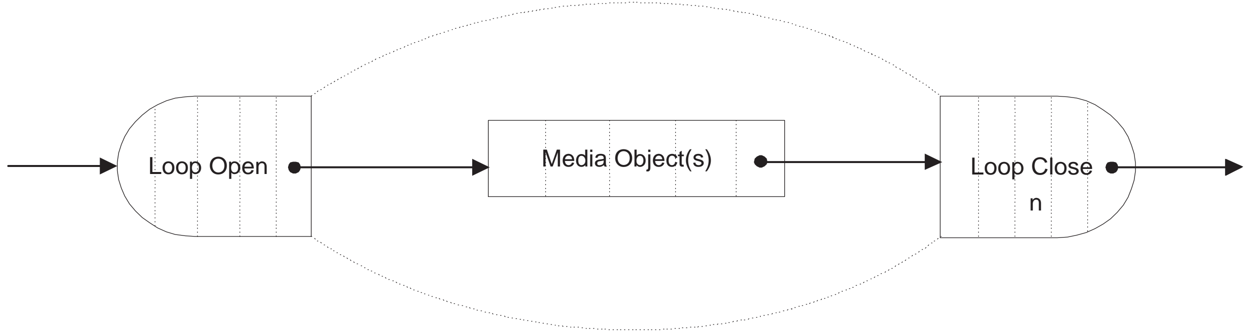

Figure 6: Loop Construct - the end of the **loop open** object enables the beginning of the appropriate media objects inside the loop. A least one loop media object must begin enable the **loop close** object, which either causes a skip back to the **loop open** object for another iteration, or enables the next object in the presentation.

the consistency checking is always going in one direction (forward or backward) the algorithm is finite. The full details of how FLIPS handles user skips are discussed in [SKD96].

3 Extended FLIPS

The loop and selection constructs defined in this section add power and convenience to the *basic* FLIPS model. Whereas the *basic* FLIPS model only recognizes a media object, the extensions define new objects to perform multi-object looping and path selection. The behavior of the extension objects is still dictated by enablers and barriers, just as are media objects. The extensions do not modify or restrict the behavior of media objects or begin and end enablers and barrier, retaining the power that *basic* FLIPS has in dealing with user interactions and variable duration media.

3.1 Loop Construct

In *basic* FLIPS, simply sending an enabler from the end of the last object of a presentation segment to the beginning of the first object will not provide looping action since only the first enabler activated for an object has any effect. A loop construct allows a presentation to specify that the playout of an object or a set of objects is to be repeated `n` times. The construct consists of two objects, a loop-open and loop-close object, as depicted in Figure 6. Both objects can be the source or destination of enablers or barriers, creating the potential for the five *basic* FLIPS states described in section 2. A loop-close object contains 2 integers, the first indicating the maximum number of loop iterations allowed by the object, and the second indicating the current iteration for the presentation in progress. Every loop-close object has a corresponding loop-open object that

Loop-open:	*Max Iteration:*	*Current Iteration:*	*State:*
L1-open	2	1	Idle

Synchronization Information

Destination	**Source**	
Begin Enablers: b_video.complete, 4w_video.complete, c_audio2.complete	*Ready Enablers:* none	*Finish Enablers:* none
Begin Barriers: none	*Ready Barriers:* none	*Finish Barriers:* none
End Enablers: self, S1.default	*In-process Enablers:* none	*Complete Enablers:* conclusion.begin
End Barriers: none	*In-process Barriers:* none	*Complete Barriers:* none

Figure 7: Loop-close Object - **L1-close** from figure 11 after one of the segments has been played and the second iteration has started but has not completed.

serves as a marker for the presentation to skip to each time the loop should be repeated. When a loop-close object transitions into the *in-process* state, the behavior of the object is as follows:

1. The current iteration is incremented and checked against the maximum allowable iterations.
2. If the maximum is equaled or exceeded, the object is immediately end-enabled, forcing a transition to the *finished* or *complete* state.
3. If the current iteration is less than the maximum, an optional GUI can be run, giving the user the choice to repeat the loop or continue forward in the presentation.
4. If the current iteration is less than the maximum and the user has not indicated otherwise, a skip is executed to the corresponding loop-open object.

Note that during the playout of a loop, the loop-close object is either in the *idle*, *ready*, or *in-process* state. The *finished* or *complete* state is not reached until the decision is made (either automatic or user-induced) to continue with the rest of the presentation. An example of a loop-close object is shown in figure 7. Similarly, once a loop has started, a loop-open object is either in the *in-process*, *finished*, or *complete* state since the object(s) that begin-enabled and freed the begin-barriers for the object remain in the *complete* state. The skip to the loop-open object performed at

the end of each iteration forces the loop-open object into the *in-process* state to maintain consistency. However, since there are no actions associated with a loop-open object, an immediate transition to the *finished* state occurs. Also, since any end barriers must have been removed from the loop-open object to allow it to enable the first iteration, any subsequent iteration will be end barrier-free, causing an automatic transition into the *complete* state when a skip to the loop-open object from the loop-close object is performed. To allow a mechanism for resetting the current iteration count for a loop, an additional consistency rule is added to all loop-open objects. The rule states that if a loop-open object is in the *idle* state, then the current iteration of the associated loop-close object must be zero. This means that when a user skips to a point in the presentation before a loop-open object, all `n` iterations of the loop will play when the loop is re-entered. Since a loop-open object is never reset to the *idle* state during normal loop playout, this rule only resets the iteration counter if a user skips to a point in the presentation before the loop-open object is enabled.

3.2 Selection Construct

A selection construct allows one of `n` presentation paths to be selected by the user and is made up of two types of objects. The first, a selection object, can be the source or destination of enablers or barriers, making it similar to the media and loop objects described previously. However, the *in-process* actions of a selection object are radically different, making use of a number of embedded path objects. Path objects are the second object type associated with a selection construct. A path object must exist for each of the `n` possible paths of a selection construct. A path object can be the source for begin and end enablers and barriers, but can only be a destination for enablers. In addition, each path object must have one enabler, termed a "run enabler", that can only come from the associated selection object. As a result of having three types of enablers and no barriers, a path object has four possible states - *inactive*, *available*, *active*, and *disabled*. The *available* and *disabled* states of a path object are "latching" states, meaning that once a path object transitions to one of these states by means of an enabler, removal of that enabler does not affect the state of the object. This behavior is different from all of the other FLIPS states, where if an enabler is removed due to a skip backwards, any state transition caused by that enabler is undone to maintain global consistency. This type of behavior allows the selection object to maintain a history of what the user has and has not seen, which is necessary for *restricted selection*. An example of this will be detailed in section 3.3. The definitions of the four path object states are as follows:

- A path object is in the *inactive* state when the object has not been begin or end enabled.
- A path object is in the *available* state when the object has been begin-enabled, but has not been end-enabled and is not currently run-enabled.
- A path object is in the *active* state when the object has been begin-enabled and is currently run-enabled.
- A path object is in the *disabled* state when the object has been end-enabled.

Note that the definitions refer to the end or beginning of the object as *has been* or *has not been* enabled. This is due to the latching nature of the *available* and *disabled* states. Once a path object has been begin-enabled which causes a transition to the *available* state, the object will not return to the *inactive* state, even if the source of the begin-enabler is forced to revoke the enabler due to a backward skip. The *active* state of a path object is the only state that requires an enabler to remain in-place. The source enablers and barriers attached to the *active* state of a path object are what cause a specific path in a presentation to play. Since the only allowable source of a run-enabler is the selection object that contains the path objects, it is the behavior of the selection object that determines which path is played. The behavior of an in-process selection object is as follows:

1. Construct the set of all path objects in the *available* state.
2. Run an optional GUI to get the user's choice of the available paths.
3. Run-enable the path object selected by the user, or the first path in the available list if nothing is selected within a time limit.
4. If no paths are available, perform default actions, which consist of a list of source enablers and barriers.

The notation for a selection construct is shown in figure 8 and an example of a selection object is shown in figure 9. If a presentation designer desires to have every path available for the user to choose from, the *ready* state of the selection object can be the source of the begin enablers for each of the path objects. Similar to the loop construct, it is useful to be able to reset the state of all of the path objects contained by a selection object. This is also done with an additional constraint, which states that if a selection object is in the *idle* state, all of the associated path objects must be in the *inactive* state. The utility of allowing only a subset of paths to be available for selection, and

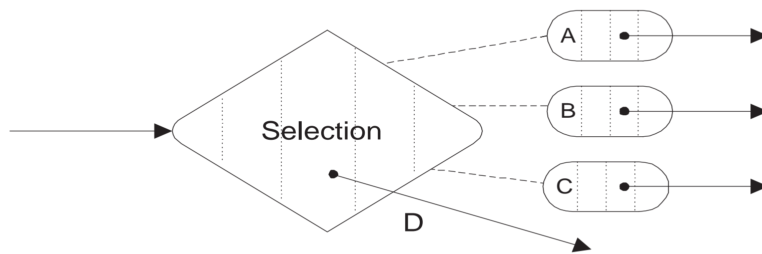

Figure 8: Selection Construct - `A`, `B`, and `C` are path objects associated with the selection object. The enabler from the *in-process* state of the selection object labeled "D" is the default action when no paths are available. Since there is always one and only one run enabler leading to each path object from the selection object, they are not explicitly shown in the diagram. Each path object is shown with an enabler leading from the *active* state to an unseen object that is assumed to be the first media object for a given path.

Path Objects: pathA, pathB, pathC	*Time Limit:* 0:30	*State:* Complete
Synchronization Information		
Destination	**Source**	
Begin Enablers: L1-open.ready	*Ready Enablers:* pathA.run, pathC.run	*Finish Enablers:* none
Begin Barriers: L1-open.complete	*Ready Barriers:* none	*Finish Barriers:* none
End Enablers: self	*In-process Enablers:*	*Complete Enablers:* none
End Barriers: none	*In-process Barriers:* none	*Complete Barriers:* none
	Default Enablers: L1-close.begin	*Default Barriers:* none

Figure 9: Selection Object - **S1** from figure 11 during the playout of one of the segments.

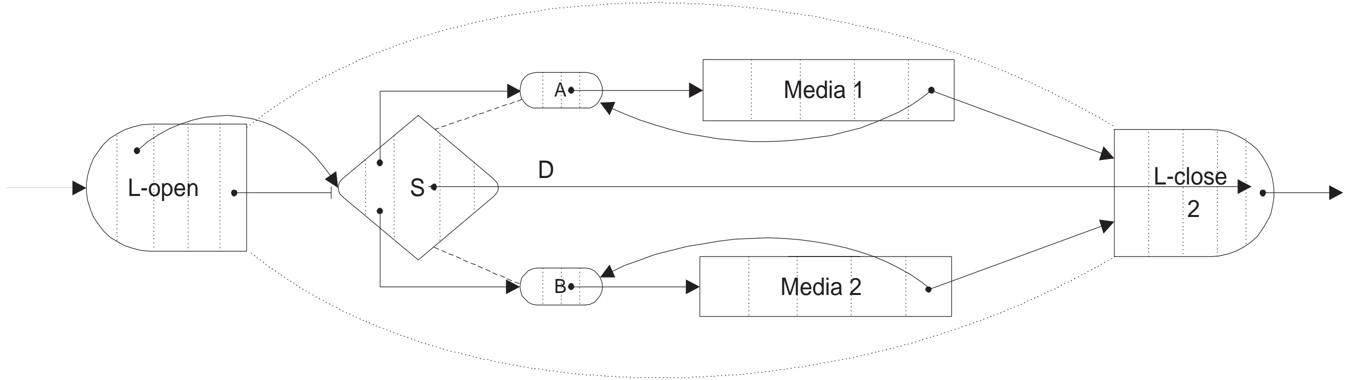

Figure 10: Simple Selection-Loop - **S** is begin-enabled from the *ready* state of **L-open** in order to prevent the available paths from reseting at the beginning of each iteration. Both path objects are begin-enabled from the *ready* state of **S**, making them both available on the first iteration. The selected path will play its media object and then begin-enable **L-close**, causing a skip back for the next iteration. The completion of the selected path will also disable its own path object, preventing the path from being selected again. On the second iteration, the unplayed path object is the only available choice and will be selected. When the second path is complete and enables **L-close**, the current iteration will equal the maximum iterations, causing a transition to *complete*. The default enabler from **S** allows the user to exit the loop without viewing another (or any) path.

providing the capability to disable a path will become clear during the discussion of selection-loops contained in the next section.

3.3 Selection-Loops

A natural combination of constructs is a selection construct contained within a loop construct. This combination allows the user to select `m` of `n` available paths for viewing, instead of just one of `n` paths. Figure 10 shows an example of a very simple selection-loop. The selection object is begin-enabled by the *ready* state of the loop-open object, but the *finish* state of the loop-open object is the source for the begin barrier. This prevents the states of the path objects from being reset at the beginning of each iteration of the loop. Both path objects are begin-enabled from the *ready* state of the selection object, making them both *available* from the first iteration. The selected path object enables the associated path when it is run-enabled by the selection object. For this simple example, each path consists of a single media object. When a path is completed, it end-enables its own path object, putting the path object in the *disabled* state. This means that during the next iteration, the path that has already been seen will not be given as an option since it is no longer available. The end of each path begin-enables the loop-close object, which causes a skip back to the loop-open object. At this point, for global consistency, all of the barriers and enablers that have

sources after the loop-open object are revoked. However, since some of the path object states are latching, the path objects can remain in an *available* or *disabled* state even when the selection object is not begin barrier-free. When the next loop iteration is started, the path object that was *active* during the last iteration transitions to *disabled*, since the run enabler from the selection object has been revoked and it has been end enabled. During the second iteration, only one path object is *available* and can be selected. Upon exit from the loop, the last path that was played remains in the *active* state since the selection object remains in the *complete* state. A user skip to back before the loop-open object will reset both the loop iteration counter and the states of the path objects. A user skip back into the loop will cause a playout of the last path selected.

A more complicated example of a selection-loop, based on the SUV on-line manual example, is shown in figure 11. Now the user has a choice of which segments to view. On the first iteration, only the basic and climate control segments are available for viewing, since these are the only paths made available from the *ready* state of the selection object. The end of the basic segment begin-enables the 4-wheel drive path, making it available for selection on the next iteration. The default action enables the end of the loop-close object, allowing a skip to the end video without viewing any more segments. The background music is not part of the selection loop, and will continue to play until the loop is exited.

3.4 Inter-Loop Conditions

Conditions that lead out of a loop require special handling to prevent the condition from being revoked every time the loop starts a new iteration. An example of a case where an inter-loop condition would be desired is if the background music of a presentation should change after 2 iterations of a loop, as shown in figure 12. An end enabler is required to end the first background music object and start the next. The latching nature of the *disabled* state of a path object is taken advantage of to provide an enabler that will not be revoked, even when the selection object reverts to the ready state upon a new loop iteration.

3.5 Other Conditions

Conditions leading into a loop, into a selection path, loops nested within another loop, or selection constructs nested within a single path of another selection construct are all situations that can be handled by the *extended* FLIPS model without any special conditions. Not all of these options may make semantic sense, but *extended* FLIPS will retain global syntactic consistency for any

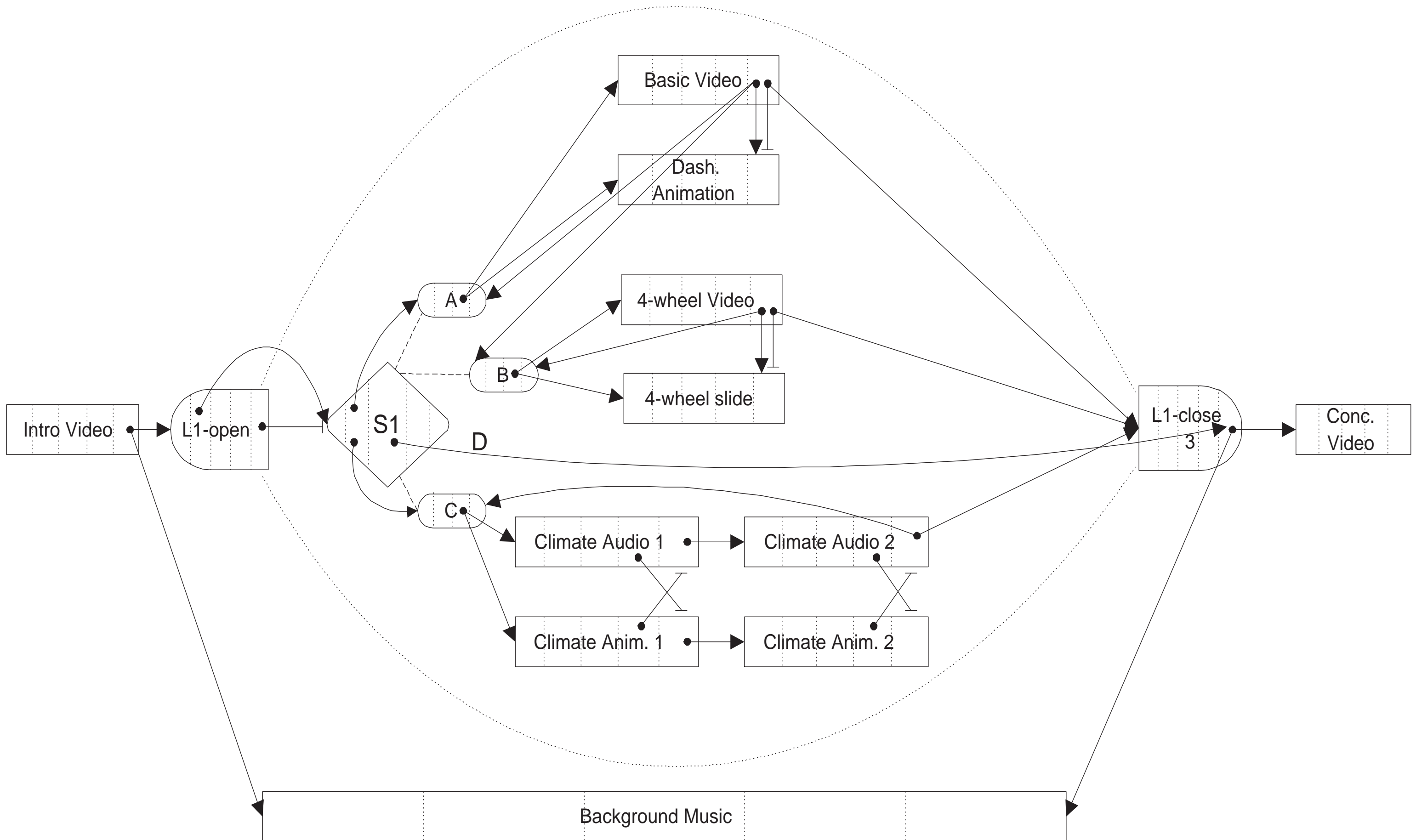

Figure 11: SUV Example - The end of the **introduction video** enables the beginning of **L1-open** and the **background music**. Like the simple selection-loop example shown in figure 10, the selection object is begin-enabled from the *ready* state of the loop-open object. Only paths **A** and **C** are available during the first iteration. The end of the last object in each path disables its own path object. The end of the **basic video** also begin-enables path object **B**, making it available during the next iteration. Once all three segments have been seen, or the default action is chosen from the selection object, **L1-close** transitions to complete, causing the **background music** to end and the **conclusion video** to begin.

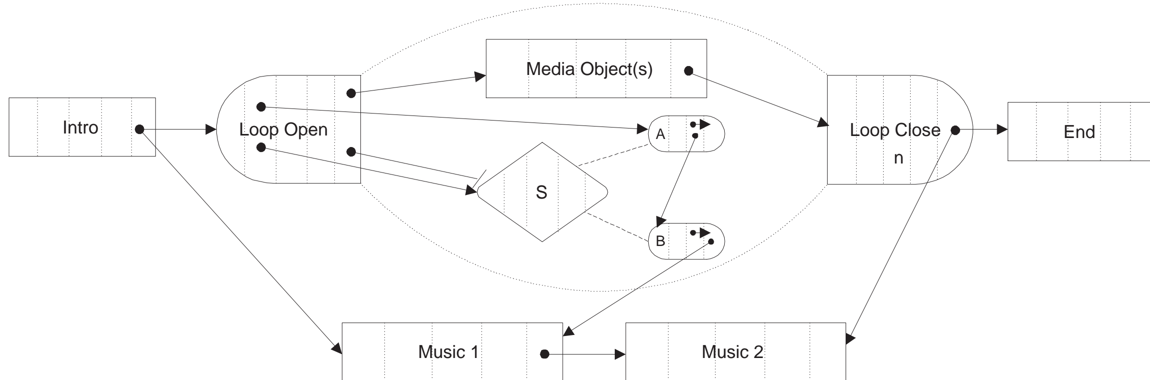

Figure 12: Condition Leading Out of Loop - path object **A** is the only path available on the first iteration and will be automatically selected. **A** begin enables **B** and disables itself, making it unavailable for subsequent iterations. On the second iteration, **B** is automatically selected and disables itself, causing the end of **music 1** to be enabled. Since the *disabled* state is latching, the end enabler of **music 1** will not be revoked during subsequent iterations. There are no default actions defined for the selection object, meaning that no events will be triggered by **S** in loop iterations after the second one.

presentation with these combinations of constructs.

4 Conclusions

This paper has extended an existing coarse-grain synchronization model, FLIPS, to include explicit path selection and multi-object looping. By combining a model that allows fully asynchronous user interactions and media objects of unknown or variable length with the ability to loop and select paths, a model has been created that can handle almost every conceivable type of coarse-grain synchronization specification. While many existing models contain some of the elements of *extended* FLIPS, no single model has the capability to efficiently handle all of the constraint types discussed in this paper.

Future work will include a full implementation of the *extended* FLIPS model and development of a graphical presentation design toolkit based on *extended* FLIPS.

References

[BHL91] G. Blakowski, J. Huebel, and U. Langrehr. Tools for specifying and executing synchronized multimedia presentations. In *2nd Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video*, Heidelberg, Germany, 1991.

[BS96] G. Blakowski and R. Steinmetz. A media synchronization survey: Reference model, specification, and case studies. *IEEE Journal on Selected Areas of Communication*, 14(1):3–35, January 1996.

[BZ92] M. Buchanan and P. Zellweger. Scheduling multimedia documents using temporal constraints. In *3rd Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1992.

[CO96] J. Courtiat and R. Oliveira. Proving temporal consistency in a new multimedia synchronization model. In *Proc. of ACM Multimedia 1996*, pages 141–152, 1996.

[CPS96] K. Candan, B. Prabhakaran, and V. Subrahmanian. Chimp: A framework for supporting multimedia document authoring and presentation. In *Proc. of ACM Multimedia 1996*, pages 329–340, 1996.

[Dra93] G. Drapeau. Synchronization in the maestro multimedia authoring environment. In *Proc. of ACM Multimedia 1993*, pages 331–339, 1993.

[LG93] T. Little and A. Ghafoor. Interval-based conceptual models for time-dependent multimedia data. *IEEE Transactions on Knowledge and Data Eng.*, 5(4):551–563, August 1993.

[Paz96] P. Pazandack. *Multimedia Language Constructs and Execution Environments for Next-Generation Interactive Applications.* PhD thesis, University of Minnesota, 1996.

[RH96] K. Rothermel and T. Helbig. Clock hierarchies: An abstraction for grouping and controlling media streams. *IEEE Journal on Selected Areas of Communication*, 14(1):174–184, January 1996.

[RS92] L. Rowe and B. Smith. A continuous media player. In *3rd Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1992.

[SKD96] J. Schnepf, J. Konstan, and D. Du. Doing flips: Flexible interactive presentation synchronization. *IEEE Journal on Selected Areas of Communication*, 14(1):114–125, January 1996.

[Ste90] R. Steinmetz. Synchronization properties in multimedia systems. *IEEE Journal on Selected Areas of Communication*, 8(3):401–412, April 1990.

[VJMB93] G. VanRossum, J. Jansen, K. Mullender, and D. Bulterman. Cmifed: A presentation system for portable hypermedia documents. In *Proc. of ACM Multimedia 1993*, pages 183–188, 1993.