The Temporal Language Component of DAMSEL

An Embedable Event-Driven Declarative Multimedia Specification Language

Paul Pazandak
Jaideep Srivastava
John V. Carlis

{pazandak|srivasta|carlis@cs.umn.edu}
Distributed Multimedia Research Center
University Of Minnesota

ABSTRACT

This paper provides an overview of the DAMSEL project, and the temporal language component of DAMSEL. The temporal component is one of three language components, and part of a framework being implemented at the University of Minnesota. DAMSEL is comprised of an embedable dynamic multimedia specification language, and supporting execution environments. The goal of DAMSEL is to explore language constructs and execution environments for next-generation interactive multimedia applications. DAMSEL supports dynamic, event-driven specifications for the retrieval, presentation, modification, analysis, and storage of multimedia data. Dynamic specifications enable system, application, and user-media interactions to affect the run-time behavior. The temporal language component of DAMSEL contains two primitives for event-driven temporal specification – supporting causation and inhibition. Specifications require (extensible) behavioral parameters to be chosen, enabling very powerful temporal relations to be defined. The other two components handle the modification, analysis, presentation and storage of multimedia data. DAMSEL components support conditional and constraint logics, enabling more complex specifications than currently possible. DAMSEL also supports an open systems view, enabling current software to be used within it's architecture.

keywords: multimedia, specification language, synchronization, event-driven, run-time event manager, embedded language, event detection

1 INTRODUCTION

The demand for, and use of multimedia has grown rapidly. One area of particular importance is the development of languages that enable programmers to easily write interactive applications which can retrieve, view, modify, analyze, and store multimedia data. DAMSEL, a **D**yn**A**mic **M**ultimedia **S**p**E**cification **L**anguage, addresses these issues using three language components and underlying execution models: the dynamic event-driven temporal component, providing the interaction-driven media and event orchestration mechanisms; the dataflow component, handling retrieval, modification, analysis and storage; and, the presentation component, handling the presentation/viewing of multimedia. At one end of the spectrum, we find proposed (and implemented) languages allowing simple static multimedia presentations to be defined. At the opposite end, we find languages, such as DAMSEL, that enable dynamic interactive multimedia applications to be created. DAMSEL is composed of just a few predicates which can be embedded within another language, eliminating the need to learn and use an entirely new language, compiler, and programming environment. The goal of DAMSEL is to explore language constructs and supporting execution environments for next-generation interactive multimedia applications. The basic features of DAMSEL include declarative specifications (strictly non-procedural), expressiveness, simplicity, conditional and constraint logics, extensible behavioral specification parameters, programming language-embedable, and an open systems approach enabling current software to be integrated. In addition, we are investigating a subclass of complex sequence-based event detection (time-

sequence detection) and the creation of a time-sequence event definition language to detect motion-based user interaction.

In section 2, we present an overview of DAMSEL, it's architecture and language components; this is followed by the details of the temporal language component in sections 3; section 4 will contain a discussion of logic support within DAMSEL, followed by a few brief examples in section 5. In section 6, we discuss in some detail, comparisons to other implementations; and, we finish with a summary in section 7.

2 OVERVIEW OF DAMSEL

DAMSEL is being embedded within C++, and includes a specification pre-processor and run-time event manager. As described above, DAMSEL has three components that provide high-level constructs for creating interactive applications capable of retrieving, modifying, analyzing, viewing and storing multimedia data (see Figure 1).

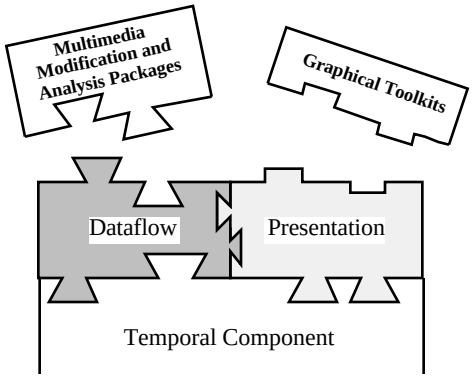

Figure 1: Overview of DAMSEL components.

The foundation of DAMSEL is the event-driven temporal language component. Specifications are defined by users and application designers to represent the behavior of the system. Each statement within a specification defines either an excitatory or inhibitory relationship between some set of events. Simply, when one event occurs, it excites (or causes) another event to occur (or inhibits another event from occurring). Since system, application and user interactions are interpreted as events, the actual behavior of the system is determined only at run-time as occurring events cause other events to be generated, based upon the pre-defined specifications. In addition, the behavior of each statement within a specification is dictated by an extensible set of behavioral parameters. It is this, in part, that enables DAMSEL to express all of the temporal relationships expressible in any of the sixteen other models we compared[7,22]. The language of the temporal component will be introduced in somewhat greater detail in section 3.

The dataflow component uses a stream model (similar to another model[21]), in which multimedia sources and sinks are specified. The media objects are modeled as continuous streams, flowing from sources to sinks. The component is used to retrieve and store data by specifying sources and sinks, which may include distributed devices, and stored or live data sources. In addition, streams can be modified or analyzed as they flow by inserting operations (image processing, filters, etc.) between the sources and sinks. The operations can be defined internal to DAMSEL, or they can be independent external processes.

The presentation component supports specifications related to the presentation of multimedia data, such as simple windows and more complex layouts. This component has a presentation server which manages the delivery of streams to complex layouts. The layouts can be defined internally, or externally using a graphical language toolkit for example. An overview of DAMSEL's execution environment is shown in Figure 2.

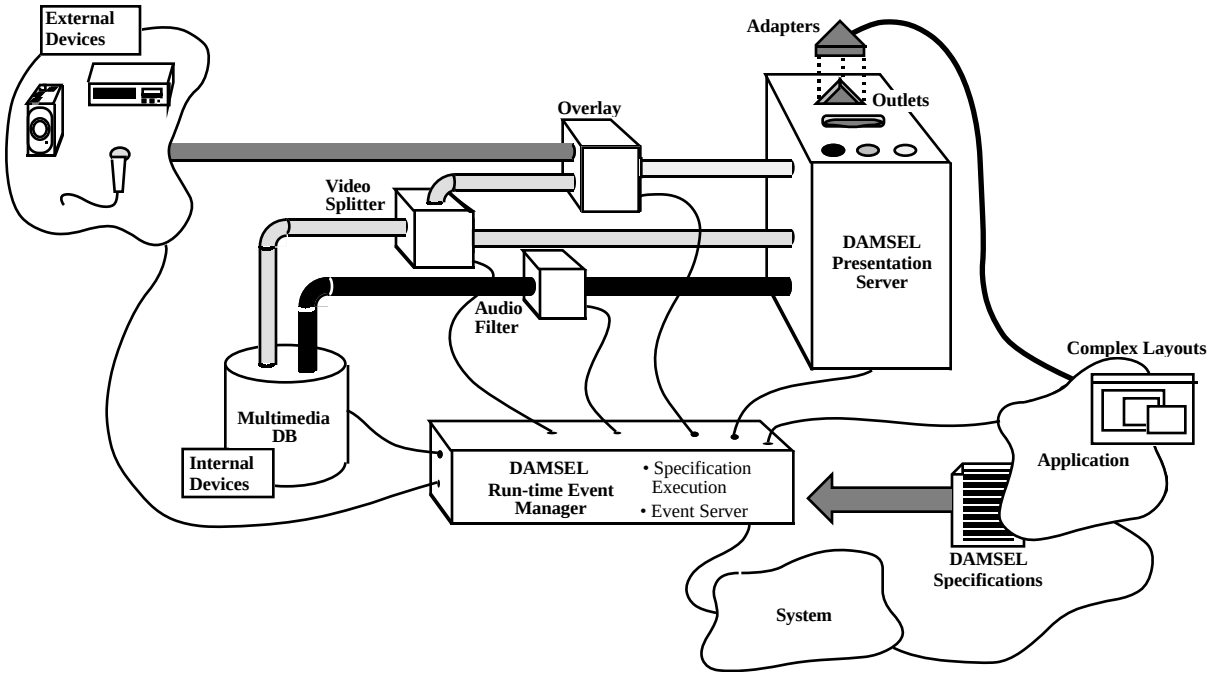

Figure 2: DAMSEL's Execution Environment

Since specifications from these last two components can be used within specifications from the timing component, which is event-driven, DAMSEL also supports dynamic dataflow and presentation specifications rather than purely static ones – therefore, user, system, and application events can affect the run-time behavior. This means that the stream definitions and presentation layouts can dynamically change due to interactions at run-time.

The specification pre-processor handles specifications embedded within C++ programs. The output generated includes the C++ programs, C++ code to support run-time execution, and the temporal, presentation and dataflow specifications in a format understood and managed by the run-time event manager/scheduler. Other implementations of event managers exist, such as *Glish*[27], but they would need to be extended to provide the necessary functionality to be used within DAMSEL's interactive multimedia application environment. DAMSEL supports C++ methods to enable users to cancel (or remove) specification statements at run-time, and to register/submit statements on the fly (at run-time).

In an effort to investigate the specification of conditionals and constraints, DAMSEL supports a basic set of temporal and causal logic. We also have defined an extensible mechanism in which conditionals and constraints may be defined within the presentation and dataflow component specifications.

3 THE TEMPORAL LANGUAGE COMPONENT OF DAMSEL

Some simple multimedia applications include presentations, in which a score is defined indicating when each media or multimedia object should be displayed on the screen. It may be a simple sequence, or a complex orchestration. (A multi-media object is an object composed of multiple data types – e.g., an audio-video object; whilst a media object is composed of a single type. We will use the terms interchangeably, and note if a distinction is required.) The notion of a score, or "temporal ordering" information, will be used by many multimedia-based applications to orchestrate the delivery of each media object. This information is generally specified by a user/programmer using a graphical interface or specification language, and then stored within the application or the multimedia objects themselves.

The set of temporal orderings is called a *temporal specification*; its structure is regulated by a temporal specification model. There is a wide range of proposed (and implemented) temporal specification models to date. At one end of the spectrum, a temporal specification model may be static, only allowing the time of delivery to be tied to a clock. At the opposite end, a model such as DAMSEL has, will support dynamic interaction-driven specifications by enabling a *behavior* of the system to be specified, while the actual execution (different for each user) will depend upon system, application, and user-media interactions (as well as resource availability and data loss, for example).

In addition, a finer level of synchronization is required to coordinate the presentations of media objects that have a high degree of *temporal interdependency*. Finer degrees of synchronization specified between media objects produce the synchronized presentation of related fragments of the media objects. An example is the synchronized playout of a video and its related audio, maintaining lip-synchronization.

Several different models supporting temporal specifications have been defined. Blakowski[6] defined three primary approaches:

- **Hierarchical.** - Using a tree structure, temporal relations are constructed using internals nodes (generally either 'parallel' or 'serial' temporal relation operators), and leaf nodes (media objects).
- **Timeline.** - Temporal relations are specified by indicating the start (and perhaps end) times of the media object presentations using implicit or explicit timelines.
- **Synchronization points.** - The temporal relations are defined by logically connecting together the synchronizing points that have been inserted within the different media objects. Each object encountering a synchronizing point waits until all objects sharing this point have also reached it (like a parallel-join).

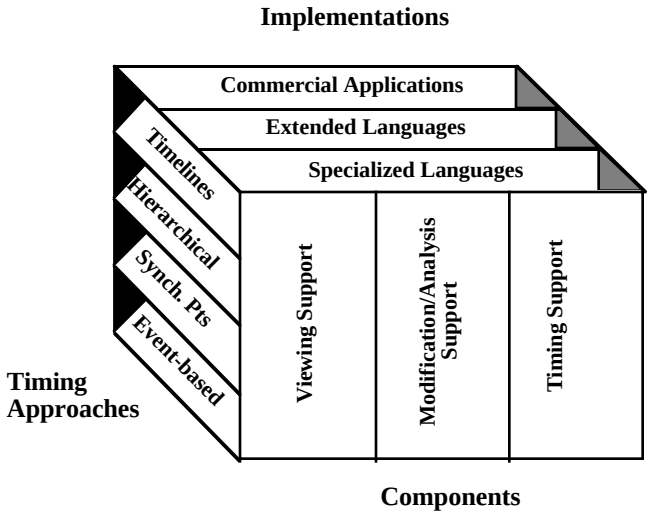

Figure 3: Current Language Approaches

More recently, several models including DAMSEL have since been developed using a fourth approach based upon events. This *event-driven* approach involves synchronizing the begin/end points (and perhaps any point in between) of a media object's presentation to the occurrence of other events. Supported events may include system events, application events, user-defined events and events associated with the execution of a media object. Figure 3 illustrates the current types of implementations, the timing approaches used, and the possible components of an implementation (which support timing, presentation, and run-time modification/analysis of multimedia data).

The temporal component of DAMSEL[22] is based upon an event-driven approach and includes two simple, yet powerful relations for expressing *activation*, *inhibition* and *fine-grain synchrony*.

3.1 Objects in DAMSEL

In our language, temporal objects are the model elements upon which relations are specified. Temporal objects include media objects, events and timepoints.

Media objects. Within an object-oriented environment, media objects may be modeled in different ways and have different meanings. Several multimedia data models have been proposed[4,8–11]. We define media objects as instances of media classes that represent both time-dependent (continuous), and time-independent (non-continuous) data types. In general, relations are defined on media objects with respect to their entire interval, called *interval specification*, or, with respect to their endpoints, called *endpoint specification*.

Current timeline models[12] and hierarchic models[4,13,14,15,21,25] use interval specification. Allen's work[16] also discusses relations on intervals within his conceptual level specification. Endpoint specification has been used in synchronization point models[2,6,24], and event-driven models like DAMSEL [1,8,17,18,20]. At a conceptual level, endpoint relations have been defined by Esch and Nagle[19]. These models will be discussed in more detail in section 6.

In addition, we classify media objects as either having a *predictable* or *unpredictable duration* [6,20]. The duration of most media objects are predictable. An unpredictable duration may involve media objects having one of the following: no inherent consumption rate, such as images; a flexible consumption rate, such as text; or, media objects that are being recorded in real-time, such as a video conference call.

Events. As stated above, the endpoints of media objects can be tied to events. Specifically, the beginning of an interval has a *begin* event, while the end of an interval has an *end* event. To provide additional flexibility, events can be defined at a *finer grain* within media and multimedia objects. For example, we could attach an event to the beginning of a particular sample associated with the start of a word within an audio clip. This is easily done when a flexible data model has been defined. *User-* and *application-defined* events are additional temporal objects that can be used within a temporal specification in DAMSEL.

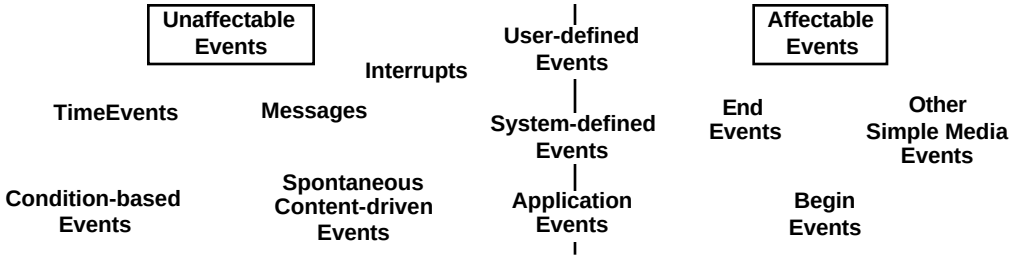

Figure 4: Event Types in DAMSEL

In addition, if image or sound analysis (and pattern-matching) is available, we may define events associated with the occurrence of a particular sound or image within the audio or video presentation. Fujikawa[17] discusses the automatic generation of events corresponding to spontaneous actions within media objects, such as video.

Using an event-driven approach, DAMSEL must characterize events as being either affectable, or unaffectable (see figure 4). Affectable events are those that the system can cause to occur (as such, methods must exist that enable the system to cause the events to occur). Therefore, events can also be associated with any method code.

There are two categories of unaffectable events, those that contain conditional logic and those that exist outside of the domain or control of the system (such as time events, interrupts, and messages). Unaffectable events that have been associated with statements containing conditional logics will occur

when the condition becomes true. For example, we can associate an event A with the simple condition of X=2 or $X > 2$ becoming true. Further, DAMSEL's temporal language component supports notions from temporal logic, so we could define an event that will become true such as when $EVTx > EVTy$ – that is, it will become true (if and) when event x occurs *after* event y has occurred. Temporal logic can be applied to events and intervals. Support for temporal logic is discussed further in section 4. It should be emphasized that any number of events (affectable or unaffectable), and simple conditions can be combined to create more complex, or *composite* events.

User, system and application-defined events may be either affectable or unaffectable. These may include events such as waiting for: a mouse-click, a specific point in time, an interrupt or message, or for a red balloon to appear in a frame of a video, all of which are unaffectable; or, causing: the closing of a window, the ending of a presentation of a video stream, or the starting of an audio stream, all of which are affectable events. In addition, events can be **instantaneous** (such as those described above), or they may have a **duration**. Events having a duration are bounded on each side by an instantaneous event, and define an interval (e.g. the interval-based event, displaying a video, is bounded by the starting of the video and its termination – which are both instantaneous events).

Timepoints. Timepoints are instants in time associated with user-defined timelines, i.e., they have no duration. Timepoints are used to specify starting and/or stopping points for media objects. We can specify a timepoint, such as t0 + 3 minutes, whose origin (t0) is one of several possible local clocks, perhaps associated with the beginning of a presentation. This is classified as a *relative time* point. We may also specify a timepoint – e.g., 12:45pm – with respect to *world time* (perhaps Greenwich Mean Time). Lastly, we can instruct the system to generate an event when some designated timepoint occurs.

3.2 Temporal Predicate Parameters in DAMSEL

DAMSEL's temporal predicates are augmented by the concepts of derivable starts, delayed starts and finishes, and behavioral descriptors:

Derivable starts. Derivable starts permit a temporal specification that only involves specifying the interval's end point – that is, when the interval will end. We then leave it up to the system scheduler to determine a proper start time for the interval's start point to satisfy this constraint. This is only applicable to intervals whose duration is known in advance.

Delays. In order to extend the flexibility of a temporal specification, DAMSEL supports delayed starts and finishes. In DAMSEL, delays may be specified as *positive* ranges, or *negative* ranges. Thus, we can express such specifications as "event x starts 30 to 33 seconds after the occurrence of event y". Ranges provide another dimension to defining delay specifications over single-valued delays. In addition, precise timing specifications using single-valued delays are generally unachievable[23].

Range delays were discussed extensively by Steinmetz[2], and later by Buchanan[20] as a means of supporting additional flexibility in specifying start and finish times. Using endpoint specifications in conjunction with range delays, one may define 48 different temporal relations (using just one of the predicates) in DAMSEL between two media objects.

We felt that negative delays should be supported as it is easy and natural to express temporal relations such as: "Start the soundtrack three minutes before the movie begins." (We can eliminate the negative delay by reversing the dependency, but the resulting statement will not be equivalent.) Negative delays are supported when the triggering or anticipated event, (e.g., the start of the movie) is predictable or derivable. As the event minus three minutes approaches (using the example above), the system continuously updates the expected occurrence of the event. At a point three minutes plus delta before the event occurs, the system generates the triggering event. In reality, it is possible that

the anticipated event may not occur due to some unforeseen problem, such as a network failure; this is equally true in real life (e.g. a movie projector breaking just before the movie should begin, while the projectionist had already started the soundtrack).

Behavioral Descriptors. Current approaches have inherently defined the behavior of their specifications. That is, the implementation (or behavior) of a specification is pre-determined. However, some flexibility of specification is possible with the restricted blocking mode defined by Steinmetz[2], which allows the programmer to specify what should occur while waiting for all streams to arrive at the next synchronization point.

The approach DAMSEL has chosen is to separate the specification from the system implementation of the specification. One parameter of every specification is then a set of zero (indicates the use of system default behaviors) or more behaviors, or programmer-defined extensions which describe how the specification should be executed. Behaviors are classified using three categories to indicate when the behaviors are executed: *activation*, *execution*, and *termination*. Behaviors are used to override the system default behaviors provided. The behaviors can be overloaded by (event) data type, and new ones can be added at any time, resulting in very flexible specifications and language extensibility. Activation behaviors specify what should happen just prior to the execution of the event. For example, they may be used to randomly generate parameters for event invocations, perhaps varying the playout of a music piece by tempo and octave. Execution behaviors can be used to control "how" the event should be executed, such as fine-grain synchronization of a video and audio stream. Termination behaviors can be used to control what should happen *after* an execution, and prior to event termination – such as repeating the event x times.

3.3 The temporal predicates in DAMSEL.

In this section we describe the predicates in the temporal language component of DAMSEL. We implement activation and fine-grain synchronization using the predicate "causes", and inhibition using "defers".

Activation. Within DAMSEL, we define causality (activation) using two events, x and y, such that "the occurrence of event x causes the occurrence of event y," where event x is the triggering event and event y is associated with some action that will be invoked. Causal relations are defined using the predicate *causes*. It takes two events, EVTx and EVTy; an optional range delay interval, r_delay (d_i,d_j); an optional set of system-defined extensions (behaviors), and a statement name, $\mathbf{s_1}$:

s_1::causes (EVTx, EVTy, r_delay, {set-of behaviors})

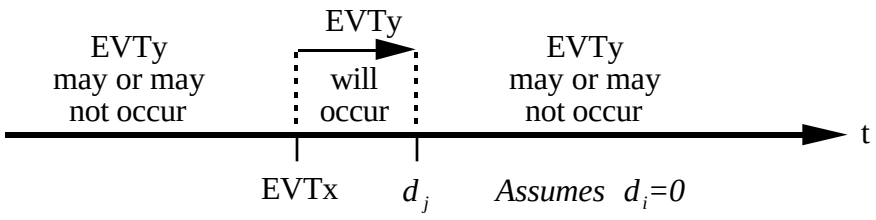

Figure 5: The nature of causes.

The basic specification should be interpreted as: "The occurrence of EVTx will cause the occurrence of EVTy. EVTy will occur at (the occurrence time of EVTx + a valid value within the range d), executed using the specified set of behaviors." Figure 5 illustrates the basic idea: prior to the occurrence of EVTx, EVTy may occur anytime by any other means; after EVTx occurs, during the interval bounded by the occurrence time of EVTx + d_i and d_j, EVTy **will** be caused to occur; after this time, EVTy may or may not occur by other means. Note that EVTx can be defined as *any* event or condition composed of

DAMSEL's conditional logic and standard C++ logical and relational expressions using global variables. In addition, EVTy can be a *set* of events.

As an example of a system-defined extension, one might pertain to the behavior of the range delay. A range delay activation behavior might characterize the delay range such that a higher priority is assigned to earlier values in the range over later ones. In this case, the system scheduler would attempt to schedule the event as early as possible within the specified range. Omitting the specification of such a behavior would enable the system default, which assigns equal priority to all values in the entire range. Remember that an event, as described in section 3.1, can be many things – providing DAMSEL with a very powerful model.

Inhibition. While activation brings about the occurrence of an event, in DAMSEL we define a means to inhibit (or defer) an event from occurring. Deferment could be thought of as an inhibitory synapse which is applied to a neuron (event) to inhibit it from firing, while causation is similar to an excitatory synapse which causes a neuron (event) to fire[26]. Basically, we define deferment using an event x and an interval t (which may be reduced to an interval of no duration, or an event), such that : "event x cannot occur *at least* until the end of t occurs." For instance, if a video is presented with background music, we may desire to have the music last (at least) as long as the video (this is a concern if the duration of the music may be shorter than the video). To do this, we defer the end event of the music *at least* until the end event of the video occurs. Note, however, that the end event of the video will not *cause* the occurrence of the end event of the music; this must be specified using causality. To be clear, deferment cannot be implemented using causality.

In DAMSEL, deferment is specified using the *defers* predicate. It takes one event, EVTy, an interval event INTa, an optional delay value, D, (default = 0), an optional set of system-defined extensions (as described earlier), and a statement name. It has the following form:

s_2::defers (INTa, EVTy, D, {set-of behaviors}).

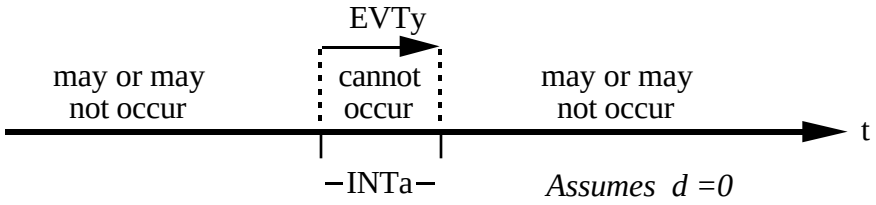

Figure 6: The nature of defers.

This specification should be interpreted as: "INTa defers EVTy"; or, not quite so terse as: "if event y would occur during interval a, the occurrence of event EVTy will be deferred *at least* until after the occurrence time of interval a's end event (+ delay value D)." Figure 6 illustrates the basic notion of **defers**: Prior to interval A, EVTy may or may not occur; during interval a, EVTy cannot occur; and, after interval A, EVTy again may occur – but *defers* will not cause it to occur (by default). As above, EVTy can be a *set* of events to be deferred. In addition, INTa can also be specified as any two bounding events, EVTa and EVTb, which describe some interval. Thus, we can interpret this statement to mean: "*when* EVTa occurs, defer EVTy *until* the occurrence time of EVTb (+ delay value D)."

The deferment of an event can have several implementation behaviors. For some events, it may be desirable, if possible, to actually disable the causing factor of the event (such as disabling a menu option). If the deferred event is an end event, by deferring it the duration of the associated interval may be affected. In the above example, we could select an execution behavior that will either play the music more slowly, extending it's duration to last as long as the video, or, repeat the music until the video ends. At the implementation level, within DAMSEL's object-oriented environment, objects may define their own methods to handle a defer-related invocation.

Fine-grain Synchronization. When we require a finer degree of synchronization between intervals than just starting and stopping at the same time, we need to be able to express this requirement to the system. Fine-grain synchronization can be used to define synchronizing relationships between intervals that are playing simultaneously. In general, fine-grain synchronization is required when fine timing relationships exist between the atoms that make up the intervals (e.g. video frames are the atoms of a video clip).

A number of different approaches exist within current models for fine-grain synchronization. Some models provide only one level of synchronization, while others provide *variable* levels of synchronization[6]. For example, 0.0 might be specified for unrelated media objects (two independent movies playing simultaneously), while 0.7 would maintain lip-synchronization, and 0.9 when synchronizing channels of high fidelity audio (where 1.0 is perfect synchronization). Some models treat fine-grain synchronization as an *extentional* relationship specified on intervals[2,6,24]. In turn, temporal relationships are actually specified between the atoms of the related intervals.

Fine-grain synchronization in DAMSEL is specified using the primitive *causes* and a system extension. We have also created an equivalent predicate, *synchs*. It takes two intervals INTa and INTb, where INTa and INTb may or may not have the same length; a synchronization factor, synch$f$, to support variable level synchronization; and, an optional set of system-defined extensions, for the behavioral specification (similar to Gibbs[21]). The *synchs* specification has the form:

synchs(INTa, INTb, synch$f$, {set-of behaviors}).

This should be read as, "synchronize the presentation of the atoms of INTa to the atoms of INTb, at least to the degree specified by synch$f$." The intervals may have different lengths, so what actually occurs at implementation is specified using system-defined extensions.) We support enumerated values for synch$f$, such as: "High Fidelity Audio", "High Definition Audio/Video", "Standard Audio", "Standard Audio/Video", etc. These values carry additional meaning that help to define second and third-order constraints at the implementation level. This approach enables the enumerated values to be mapped to any underlying implementation. It also provides a suitable level of abstraction, allowing the user/programmer to select a value without having to know *how* it will be implemented.

3.4 Specification Conflict Resolution.

Conflicts within a specification may occur – for example, one statement may specify to cause an event x some at time between t_2 and t_4, while another statement specifies that event x should be deferred from t_0 through t_3 (the specification for this situation is below). Within any approach which supports both activation and inhibition, conflicts within a specification may result. We can resolve some conflicts at compile-time using *static conflict resolution*, when specification statements are attached to some timeline (global or local clocks). However, within a dynamic event-driven system, statements may not be attached to a timeline, but only to the occurrence of other events. In these situations, conflicts must be resolved at run-time using *dynamic conflict resolution*.

causes (event(time(t_2)), EVTx, (0,2*unitTime));
defers (interval(time(t_0),time(t_3)), EVTx);

Static Conflict Resolution. Static conflict resolution requires that some rules be defined to resolve conflicts at compile time. Figure 7 illustrates the example introduced above which has a conflict that needs to be resolved (note that many possible conflicts are possible). In this example, one statement defers event x, while another causes event x during intervals that overlap. Several possible resolution policies may be used, such as truncating one or the other interval (by modifying the specifications), so no overlap occurs; or, perhaps eliminating one of the statements. Conflict resolution policies may be specified within a rule-base system using simple precedence order, or more complex hierarchical-based approaches.

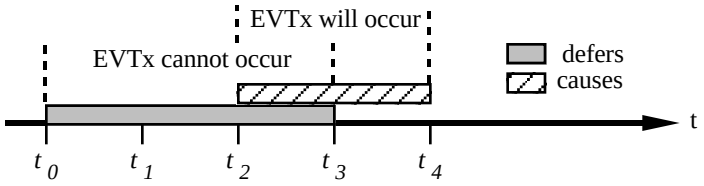

Figure 7: Static Conflict.

Dynamic Conflict Resolution. Dynamic conflict resolution is required when the conflicts cannot be detected at run-time. An example of one possible run-time conflict is shown in figure 8, and it's specification is below. The first statement specifies that event x should occur sometime between [the occurrence time of event z + 0 secs] and [the occurrence time of event z + 300 secs].

causes (EVTz, EVTx, (0,300));
defers (interval(MovieM), EVTx);

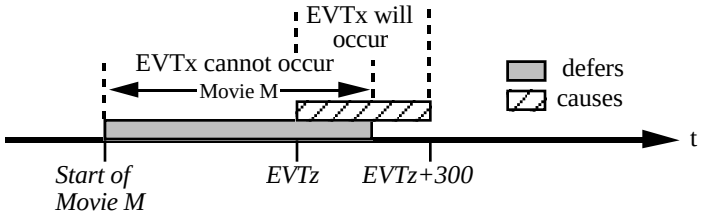

Figure 8: Dynamic Conflict.

The second statement specifies that while *Movie M* is playing, event x cannot occur. Whenever a *causes* and *defers* interval overlap, there will be a conflict. In this example, if event x were to occur after the movie, then no conflict would occur. If it occurred during the movie, then conflict resolution is required – such as, not allowing event x to occur at all; or, causing event x to occur once the interval associated with the movie has ended. Dynamic conflict resolution is specified using behaviors. The first resolution is the default – if an event is deferred, and another statement attempts to cause it to occur, nothing will happen (completely inhibited). In the second resolution, we simply add a termination behavior, *postCause()* that will cause event x *if* it would have occurred during the deferred interval. So, the modified *defers* statement is below:

defers (interval(MovieM), EVTx, postCause());

4 LOGIC IN DAMSEL.

Within event-driven systems what actually occurs at run-time generally cannot be pre-determined. For this reason, it is necessary to be able to constrain and conditionally test the state of the system. Therefore, we have chosen to support the specification of constraints and conditionals within DAMSEL. Since temporal and causal relationships are intrinsic to the temporal model, we have borrowed ideas from temporal and causal logic enabling us to study implementation mechanisms.

In addition, since the design of DAMSEL emphasizes an open systems approach, there will be components that will exist outside of DAMSEL's control yet interact with it. This will include external processes used in the dataflow component, and graphical interfaces used in the presentation component. The languages and designs used by these systems may vary widely, causing a language barrier. Also, the constraints and conditions applicable to the myriad of systems which may interact with DAMSEL would be too difficult to foresee. Therefore, to support conditional and constraint specifications involving an

open system architecture, we have defined an extensible mechanism that can be used to support constraint enforcement and conditional logic based upon the state of the external processes. We will describe the intrinsic conditional and constraint support first, followed by the extensible extrinsic support (see Figure 9).

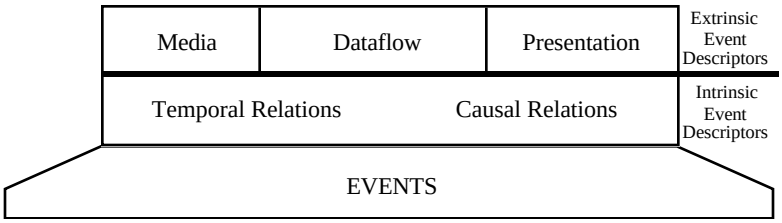

Figure 9: Intrinsic and extrinsic conditionals and constraints support.

4.1 Intrinsic conditional and constraint logic support.

Conditional temporal logic can be used, for example, to determine the current state of the system, and to test the order in which things occurred. As an example, we may be interested in determining if event x occurred *before* event y, which may be expressed as: (EVTx < EVTy). Or, we may want to determine if event x occurred *during?* interval A. DAMSEL supports *before?*, *after?*, and *concurrent?* operators for events, and the same set plus *during?* and *overlaps?* for conditionals involving intervals. Each allows a range variable to be specified, such as event x is before? (<?) event y *by at least 60 seconds*:

$$(\text{EVTx} <? \text{EVTy} , \leq 60)$$

In addition, the condition can indicate specific occurrences of an event: any occurrence, the Nth occurrence, or the first N occurrences. However, currently we have not designed any history-tracking mechanisms (i.e., event traces), so conditionals which refer to specific occurrences are only valid while those occurrences are current. (Thus, if EVTy and EVTx have both occurred for the tenth time, we cannot now test if the third occurrence of EVTy followed the fifth occurrence of EVTx, since no history is kept.)

More complex conditionals may be defined using the conjunctions, "&" (and), and "|" (or). Using the operators above, one will be able to specify conditions representing all of Allen's 13 interval relations[16], in addition to relations between intervals and events, and between events. We can use these conditionals in triggering events within a *causes* statement. Conditions may also be used within *defers* statements to defer events while the condition is true.

The definition and support of a causal logic is more involved. The general notion of a causal logic for this domain will allow us to be able to conditionally test the direct cause of an event using *caused?* and *causedBy?*. As such, we can define a condition such as, "EVTx → EVTy." This condition would be true when event x caused event y to occur.

Finally, we may also want to constrain the system by defining temporal and causal constraints. The system scheduler will use them at run-time to ensure that the constraints are met. Examples of a few temporal constraints include: "event x cannot occur after event y," and "event x cannot occur before interval A." Currently, the temporal constraint operators we have defined include *before* and *after*, with additional parameters as defined above. Using them, we could restrict events to occur before or after the occurrence of another event. The causal constraint logic includes *cannotCause* and *cannotBeCausedBy*. Although the logics are not complete, we are more interested in the general notion and mechanisms by which such conditionals and constraints can be managed by the system.

4.2 Extrinsic conditional and constraint support.

Since external processes are outside of DAMSEL, we have adopted a reactive systems approach to support conditionals and constraints. In these processes, events will occur before DAMSEL can know about them, so they cannot be constrained from occurring. Instead, updating events will be sent to the run-time event manager after they occur, allowing it to react as defined in it's specifications. In addition, conditionals cannot be formulated which refer to the state of external processes, since their variables are not local to DAMSEL. To enable this, any updates to the state of external processes can be sent as events to the run-time event manager. Specifications can refer to the values contained in the event objects within conditionals.

Firstly, to define reactive constraints – those which are executed after a constraint has been violated – we define specialized event classes within DAMSEL (the base class hierarchy of events includes two classes, "affectable_events" and "unaffectable_events", to which subclasses may be added). Within these new classes, we define the attribute descriptors that will be needed for constraint violation detection, and conditional formulation. For example, if we want to constrain window movement within our application, we would use the window event class – one of the basic extension classes we are defining. The definition of this class includes attribute descriptors relevant to a window, such as window location, size, and visibility. When window "A" is moved within the user interface layout, that process will send a window(A) event to the run-time event manager. The constraints for this event will be checked with the associated instance within the specialized class for window "A"; and, if a violation has occurred, the system will react. It may move the window back (a compensating event), or perhaps advise the user of the violation and ask him/her if the constraint should be overridden. The resulting behavior will depend upon the constraint methods specified.

Conditionals may be formulated using attributes from these specialized classes. The attributes can be updated at any time by having the external process send an update event.

5 A FEW EXAMPLES.

DAMSEL has few predicates, and by simply using defaults (e.g. behavior, timing) it is easy to define very expressive specifications. It is also possible to define very complex specifications, with the ability to hide much of the complexity. By encapsulating the complexity within new behaviors, this complexity is hidden from the user as is simply viewed as one more extension. In this section, we illustrate some basic DAMSEL specifications.

To simply specify that a media object (training video) should be played, only one statement is required:

causes (someTriggeringEvent, event(TrainingVideo1.start()))

This will start the video once *someTriggeringEvent* has occurred. To embed this within C++ code, so that it can be executed within sequentially executed code, simply remove the triggering event, and use the embedded **causes** syntax, by preceding it with an underscore (the video starts once the statement is executed):

_causes (, event(TrainingVideo1.start()))

If the media object is not connected to a datastream (using predicates from the dataflow component), it will be presented using a default (or pre-defined) application for that media type (or media object) on the local host machine.

Next, we described a bit more involved example that defines a slide/audio presentation. One initial event, *beginShow*, activates the presentation which displays slide_1 and plays audio_1. After audio_1 has

ended, we want to leave up the slide for the user to review, or while some discussion ensues. After our selected time limit of about one minute, we want to continue so that the presentation doesn't run long. This requires us to end the current slide, and start the next slide and audio. One restriction is that the slide cannot be ended *at least* until the audio has finished (the user might try to end the slide through the user interface). However, once the audio has finished, the user can move on to the next slide at any point – it will occur automatically after 60 seconds. An illustration of this scenario is in figure 10, and the specification required follows.

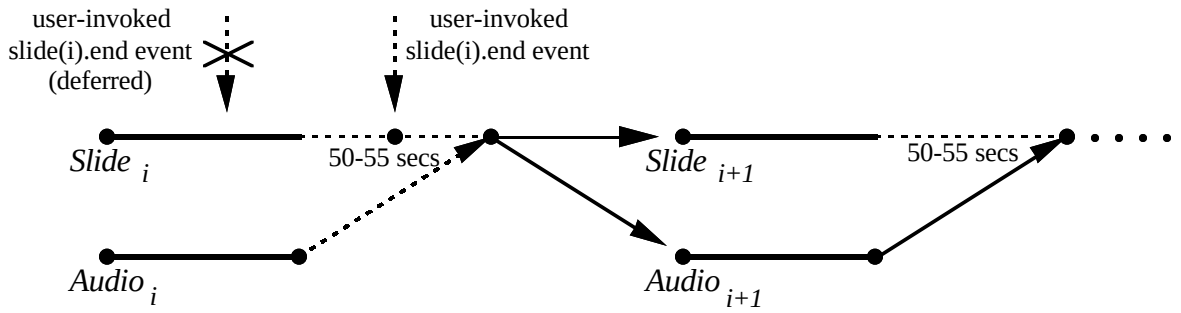

Figure 10: Temporal Specification Example: A slide show.

0. **causes** (beginShow, (slide1.start, audio1.start));

1. **causes** (audio_i.end, slide_i.end, (60,62));
2. **defers** (audio_i, slide_i.end);
3. **causes** (slide_i.end, (slide_{i+1}.start, audio_{i+1}.start));

Statement 0 activates the presentation; statement 1 will cause the end of the slide 60-62 seconds after the audio ends (after the audio_i.end event occurs). Statement 2 inhibits a slide_i.end event from occurring while the audio is playing; and, the last statement activates the next slide and audio.

For every additional slide, statements 1-3 will have to be repeated. However, it would be straightforward to create a method to handle a slide presentation like the one above – acting as a macro – and, could look something like this (where "slide" and "audio" are the name prefixes for the **10** media objects to be presented):

causes (beginShow, event(slidePresentation("slide", "audio",10)));

The next example describes an interactive examination, illustrated in figure 11. In this short example, a video is played and then the user is tested based upon its content. The video has accompanying audio tracks, both in german and english. The test is accomplished using slides which the user must answer. Wrong answers cause a related segment of the video to be re-played, so the user can see where the correct answer could have been found. After the user has seen the review (in slow-motion, with a magnified view of the area of interest), the next test slide is presented. Correct answers simply cause the next test slide to be presented. For some variety we have added background music, which will play until the video and test have completed. The following text describes the statements required (we have chosen one of several possible ways to implement this example). This example uses language constructs from all three components[28] of DAMSEL to provide an overall example.

The following three statements are used to play the video, audio, and music. *tv1* is the name of the test video; *eng-tv1* and *ger-tv1* are the names of the english and german soundtracks; and, *music-tv1* is the name of the music soundtrack. In the first statement, a starting event *startTest*, causes the test video and audio to begin. *synch*, the execution behavior used, specifies that these objects should be played using "Audio-Video" quality fine-grain synchronization. The second statement starts the music. Since we don't know how long the music is we simply defer the end event of the music until at least until an *exitEvent* is generated, signalling the end of the test. For additional variety, we have defined

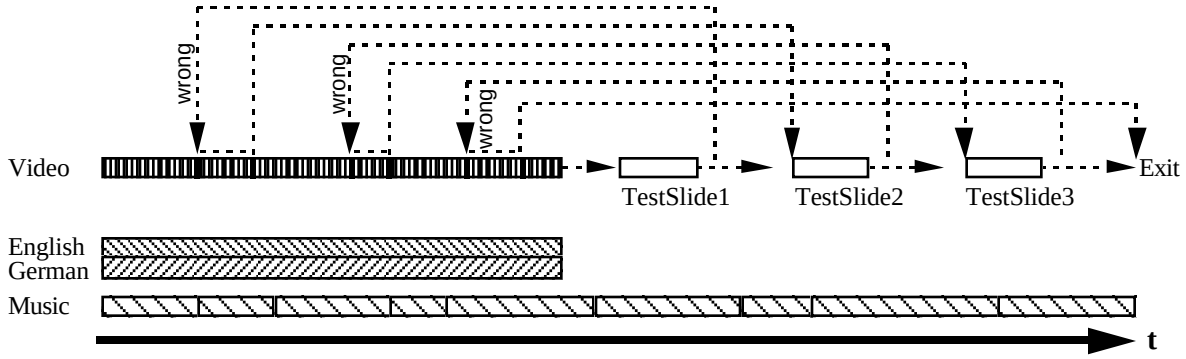

Figure 11: A video testing example.

an activation behavior that randomly selects the tempo and octave for the digital music, each time it begins. A termination behavior, *repeat()*, simply repeats the music as long as the end event is deferred.

causes (startTest, (tv1.start, eng-tv1.start, ger-tv1.start), ,synch("Audio-Video"))
causes (startTest, music-tv1.start)
defers (tv1, music-tv1.end, 0, (randomizeMusic(tempo, octave), repeat()))

The following set of statements are used to control the test itself. The first statement on the left causes the first test slide, *tslide1*, to be presented; the following statement sets the video rate to 0.3 of its normal rate so if it is played again, it will be played in slow-motion.

The set of four statements on the right are used for *each* slide to handle correct answers and wrongs answers. If a correct answer is given (the application validates and generates either a $tslide_i\_correct$ or $tslide_i\_wrong$ event), then the first statement ends the slide, and the following statement starts the next slide. If the answer was wrong, the user is shown a re-play of the video, accompanied with a magnified view of the area of interest in the video. Each re-played segment of the video is marked with two bounding events, $replay_i.start$ and $replay_i.end$. (As part of the execution of the $replay_i.start$ event, it assigns the correct area to be magnified within the *zoom* stream object described later.)

causes (tv1.end, tslide1.start)
causes (tslide1.start, event (tv1.rate(0.30)))

causes ($tslide_i$_correct, $tslide_i$.end)
causes ($tslide_i$.end, $tslide_{i+1}$.start)
causes ($tslide_i$_wrong, $replay_i$.start)
causes ($replay_i$.end, $tslide_i$.end)

The next set of statements defines the dataflow for this example. First, the stream objects are defined (on the left); then, the dataflow connections are made (on the right). This dataflow is illustrated in figure 12. We have not connected the music to the server (again, just for variety), so it will play on the local host machine.

videoZoom = new (videoMagnifier);
splitter = new (videoSplitter);
server = new (presentationServer);

vt1 → splitter();
splitter(0) → videoZoom(0);
videoZoom(0) → server(0);
splitter(1) → server(1);
eng-tv1 → server(2);
ger-tv1 → server(3);

Finally, the outlets and adapters are defined. We have created both german and english adapters, indicating which streams should be connected to each. The adapter, *videoTest*, may connect to either outlet, and do so at run-time using either of the last two statements.

createOutlet ("ENGLISH", mpeg:0, mpeg:1, audio:2);
createOutlet ("GERMAN", mpeg:0, mpeg:1, audio:3);
createAdapter ("videoTest", mpeg:video, mpeg:video, audio:audio);
connect ("videoTest", "ENGLISH");
connect ("videoTest", "GERMAN");

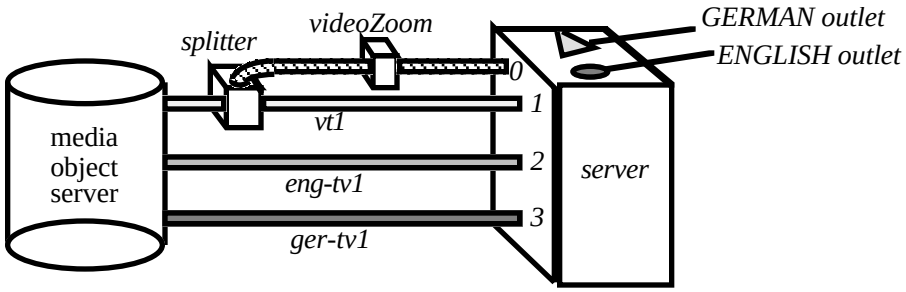

Figure 12: Dataflow for the video testing example.

This basic example illustrated the use of all the components of DAMSEL. The example could be extended in many ways, including embedding parts of it within C++ code and executing statements on the fly. Some people may be more comfortable using the specification language and adding extensions, while others may prefer to use it within C++. Finally, at the user level, we expect that easy-to-use graphical interfaces should be used; although for simple specifications it probably isn't necessary.

6 COMPARISON TO RELATED WORK.

As we said in the first section, the richness (and expressiveness) of a language or approach is directly associated with the set of objects and types of relations that can be expressed. This section briefly describes efforts in the area of temporal models and compares them to DAMSEL. A comparison covering all three components can be found here[7,28].

In section 3, we described the four current timing approaches. Here we will first provide an overview of several projects, and then compare them to DAMSEL.

6.1 Current models.

Timeline models. Timeline models provide very straightforward capabilities to define specifications, and in general, use very intuitive graphical interfaces for a presentation's specification. These approaches probably evolved from an applied, rather than theoretical basis. The simplicity of the model generally discounts the need for validation of specifications. All timeline models are restricted by the number of temporal relations that may be defined (to the 13 interval relations defined by Allen[16]).

Drapeau and Greenfield[12] defined a specification language called MAEstro using a timeline approach. This work includes a timeline editor in which media intervals are placed on the timeline, indicating the start and stopping times. The language is purely graphical in which only static specifications can be defined. This model does not support fine-grain synchronization.

Hierarchical models. Hierarchical models have a more theoretical/mathematical foundation, and their specifications are also relatively straightforward since generally only two operators are available – this approach can easily be implemented within a programming language. The model also is restricted in the number of temporal relations that may be specified since it only supports interval (object-level) specification.

T.D.C. Little[14] has extended the basic model by adding delays when using the parallel operator which allows a few additional temporal relations to be specified. He has discussed storage of the specifications using database relation tables, and includes a discussion on support for playing multimedia in reverse.

This model does not support fine-grain synchronization.

Notably, Gibbs, Breiteneder and Tsichritzis [21] have defined an object-oriented framework to implement a hierarchic approach supporting temporal specifications ("temporal composition"). A specification is defined using object instantiations and method invocations. Although they have mentioned the use of events, implementation details were not discussed. Their specifications are also static; although the conditional execution of code within the programming language used should enable selective specification execution. This model does support variable levels of fine-grain synchronization.

Hamakawa, et. al.[13], have defined a graphical notation using a concept borrowed from LateX to define a 'temporal' glue in their implementation of a hierarchical model. To define hierarchies, composite multimedia objects are constructed. The composite objects are then scheduled using a relative timeline. The glue provides additional flexibility when deriving a temporal layout of the specification. This model does not support fine-grain synchronization.

Wijesekera, D.K.Hosekote, and Srivastava[15] also have introduced a hierarchical model. The model implements delays by inserting null intervals of some duration. This model, in addition, has extensively focused on fine-grain synchronization between master and slave channels.

Schloss and Wynblatt[4] describe a multi-layered multimedia data model, and a temporal event calculus. The key temporal operators include concatenate, and overlay (serial and parallel), while fine-grain synchronization is supported using synchronization points. Specifications are stored within temporal structures in the data model.

Synchronization point models. The synchronization point approach evolved out of synchronization techniques of operating systems and parallel programming languages further extended by specific requirements for handling multimedia data[2]. One advantage of this model over the previous ones is that it naturally supports both coarse and fine-grain synchronization using the same synchronization point paradigm.

Steinmetz[2] introduced the synchronization point model. This model (based upon endpoint specifications) made possible several additional relationships that could not be defined using the above (interval-based) models. His approach defined a complex statement to extend programming languages to support synchronization point specifications (similar to a complex SQL SELECT statement). In addition, he introduced the concept of alternate activities, which are individually-specified actions to be executed once a multimedia stream reaches it's synchronization point, and while it is waiting for other designated streams to reach that point. He also was the first to address range delays, and alternate activities when exceptions occurred.

Blakowski, Huebel and Langrehr[5] extended Steinmetz's model by adding support for unpredictable durations, timers (to support time delays and timeline type specifications), and interactive objects which are user-driven events. In addition, he discussed possible extensions to his model, including: waiting actions, acceleration actions, skipping actions and alternate presentations to handle resource constraints.

Schnepf, Konstan, and Du[24] defined a presentation approach using synchronization points at the implementation level and events within a two-predicate specification language (they also define an extended timeline-based graphical interface). Their work supports alternate activities and fine-grain synchronization, implemented basically in the same way as Steinmetz above. In addition, they described support for maintaining synchronization when skipping forward and backward through a specification; and, they also considered spatial issues, such as window dependencies and window overlapping conflicts in four dimensions. They have also defined additional synchronization point semantics using "barriers" that can be applied to media objects.

Event-based causality models. This most recent approach has been chosen by several research projects, and has proven thus far to be capable of providing the most flexible synchronization primitives

of any approach.

The work of Horn and Stefani[18], and Blair, et. al.[1], are similar in that they both evolved out of work on Esterel, a real-time synchronous specification language project. The language includes the two operators, parallel and serial, but the number of the temporal relations that can be specified is achieved using synchronization based upon sending and waiting for signals (or events). These signals could conceivably be sent because of user interaction, or system and application-level events.

There are a few differences between these projects. The first project (Horn, et. al.) supported time as a valid event, and minimum time point-based delays. The second project defined execution relative time to support a timeline model, range delays, and system-level support for fine-grain synchronization.

Vazirgiannis and Mourlas[8] have defined a script-based approach which supports parallel- first, parallel-last, sequential and repetition operators, and events. It has been implemented using an object- oriented model and supports the composition of multimedia data (composite objects). The authors have focused on temporal and spatial specifications in an attempt to remain platform independent (believing that "configurational" or modification specifications will be platform dependent). In addition, they have defined n-ary and unary actions to be executed when objects are synchronized. Unary actions, applied to individual objects, include operations such as playing, suspending, cropping or scaling; while n-ary actions, applied to a group of objects, might include overlapping, grouping and fine-grain synchronization. Finally, a specification can also include exception handling and interrupt handling routines.

Buchanan and Zellweger[20] defined an event-based specification language. Their work included support for range delays with minimum, maximum and optimal values as well as cost measures for stretching or shrinking the execution duration of a media object being played out. In addition, the Firefly project included validation mechanisms to check a specification both at compile-time and at run-time for inconsistencies. No support for fine-grain synchronization is described. They support event deactivation by allowing some types of events to be turned on or off. In addition, they support object behaviors (but not specification behaviors). The language of their model was not introduced, however a useful graphical notation to support limited event-based specifications was. Their approach is restricted to to temporal specifications.

Fujikawa, et. al.[17] have defined a hypermedia-based approach which supports temporal synchronization using events, and parallel and serial operators using timepoint delays. They have also allowed synchronization to be tied to the "first to finish", or "last to finish" when a group of objects is specified. The work does not address fine-grain synchronization.

6.2 Model comparisons.

Of the approaches above, the models supporting the timeline approach cannot handle unpredictable durations, and their specifications are static. All of the hierarchic models should be able to support delayed starts, and unpredictable durations. However, this approach is restricted in the types of temporal relations that can be defined, as discussed by Blakowski[6].

End point specification models, which include both synchronization point and event-driven approaches, are more powerful than the previous two approaches because (at a minimum) they can define more temporal relations. The synchronization point approach allows media objects to be temporally "tied" together by defining synchronization points anywhere within the media objects; thus, they can handle media objects of unpredictable duration. Two of them[6,24] allow user-interaction events to be used as synchronization points, while none support user- or application-defined events (such as "x==6"). All three support positive delayed starts (Blakowski[6] and Schnepf, et. al.[24] support single-valued delays, while Steinmetz[2] allows range delays); waiting actions (describes what to do while waiting for other media objects to reach a synchronization point); and, fine-grain synchronization using extensional relations.

Event-driven approaches are more powerful than synchronization point models since they can also define temporal relations between media objects and other types of events, including system, user, and

application events. This enables the occurrence of these events to affect the behavior of the execution. Horn and Stefani[18], and Blair[1], use the programming language Esterel, which provides the strengths of a high level language; while on the downside, requires users and programmers to learn an entire language. Unlike most implementations, DAMSEL uses a declarative language design, which is easier to read, write and validate than specifications written using a procedural language[23]. In comparison to DAMSEL, none of models discussed support a general form of deferment, behavioral specifications, or temporal and causal logics.

In addition, run-time resource management is a factor in any multi-user environment, and few implementations have addressed this. Using one approach, if the resources specified were not available, the specification would not play[21]. Another approach described support for alternate presentations if some set of resources were low or not available[6]. In DAMSEL, the use of resource-related behaviors and conditional specifications are two ways to provide resource-sensitive execution.

DAMSEL is the first project to date that has focused on the definition and integration of components to support dynamic specifications for the timing, presentation, and modification/analysis of multimedia data. In addition, it introduces support for deferment, and conditional and constraint logics, and extensible behaviors.

7 SUMMARY

Within this paper we have presented an overview of the temporal language component of DAMSEL, a dynamic multimedia specification language embedded in C++. Specifications in DAMSEL are dynamic, since they are event-driven. This means that system, application, and user-media events can be used within the specifications enabling very dynamic and interactive applications to be defined. We have also introduced several new concepts and ideas to make the language more powerful and useful: deferment, negative range delays, behavioral extensions, temporal and causal relations, and conditional and constraint logics. In addition, the language is simple, expressive, and extensible. DAMSEL supports an integrated approach, the separation of specification from implementation, and it's embedable so that one may take advantage of the power of a high-level programming language.

We are also working on complex event detection – particularly, sequence-based event detection, which is not currently addressed by any multimedia language projects known to us. This entails the definition of a language to describe the sequences to be detected, and supporting mechanisms. Other interesting areas may include the development of history-tracking mechanisms to support playback, playing in reverse, and skipping. Since the behavior of the system is event-driven, these issues are not as straightforward as in other models. In addition, security in multimedia has not been addressed with regard to specifications. It should be possible to define authorizations on media objects and events to restrict access, and therefore restrict the behavior of the system based also upon access privileges. If useful, it would also be possible to define activation *levels*, such that an event would have to be excited (triggered) i number of times before it actually fired.

Eventually, advanced multimedia languages and applications may support open systems architectures, as exemplified by the ISO/IEC PREMO standard in progress[3], and demonstrated to an extent by the MAEstro project[12]. We are designing DAMSEL with this in mind.

It is our hope that the ideas introduced and demonstrated in DAMSEL will be incorporated within next-generation systems, thereby providing more sophisticated capabilities than currently possible. Next-generation applications, such as scientific analysis and simulation, and interactive multimedia will require more powerful multimedia languages and applications than are currently available today.

The DAMSEL language is being implemented in a UNIX environment in the Distributed Multimedia Center, Computer Science Department, University of Minnesota.

8 ACKNOWLEDGEMENTS

This work has been funded in part by NIST (National Institute of Standards and Technology). We would also like to thank Prof. J. Konstan for his time and valuable comments on some of the preliminary work leading to this paper.

9 REFERENCES

[1] Blair, G., et al., "An Integrated Platform and Computational Model for Open Distributed Multimedia Applications," in *Network and Operating Systems Support for Digital Audio and Video. Third Int'l Workshop Proceedings*. 1992. Germany. : p. 223-236.

[2] Steinmetz, R., "Synchronization Properties in Multimedia Systems," *IEEE Journal on Selected Areas in Communications*, 1990. 8(3): p. 401-412.

[3] ISO/IEC JTC1/SC24, *Information Processing Systems — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO)*, ref 14478-1,2,3,4, 1994.

[4] Schloss, G. and M. Wynblatt, "Building Temporal Structures in a Layered Multimedia Data Model," in *ACM Multimedia 94.*

[5] Blakowski, G., J. Huebel, and U. Langrehr. "Tools for specifying and executing synchronized Multimedia presentations," in *2nd Int'l Workshop on Network and Operating system support for Digital Audio and Video.* 1991. Heidelberg, Germany.

[6] Blakowski, G., "Tool Support for the Synchronization and Presentation of Distributed Multimedia," *Computer Communications*, 1992. 15(10): p. 611-618.

[7] Pazandak, P. and J. Srivastava, "A Multimedia Temporal Specification Model Framework and Survey," University of Minnesota. Technical Report in progress.

[8] Vazirgiannis, M. and C. Mourlas, "An Object Oriented Model for Interactive Multimedia Presentations," *The Computer Journal*, 1993. 36(1): p. 78-86.

[9] Mano, "Object Model Facilities for Multimedia Data Types," 1990, *GTE Technical Report.*

[10] Gupta, A., T.E. Weymouth, and R. Jain. "An Extended Object-Oriented Data Model For Large Image Bases," in *SSD*, 1991, Zurich.

[11] Ishikawa, H. and et. al., "The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System," *ACM TODS*, 1993. 18(March): p. 1-50.

[12] Drapeau, G.D. and H. Greenfield. "MAEstro - A Distributed Multimedia Authoring Environment," in *USENIX.* 1991. Nashville, TN.

[13] Hamakawa, R., H. Sakagami, and J. Rekimoto. "Audio and Video Extensions to Graphical Interface Toolkits," in *Network and Operating Systems Support for Digital Audio and Video. Third Int'l Workshop Proceedings.* 1992. Germany.

[14] Little, T.D.C. and A. Ghafoor, "Interval-based Conceptual Models for Time Dependent Multimedia Data," *IEEE Trans. on Knowledge and Data Engineering*, 1993. 5(4): p. 551-563.

[15] Wijesekera, D., D. Kenchamanna-Hosekote, and J. Srivastava, "Specification, Verification and Translation of Multimedia Compositions," Technical Report 94-1, University of Minnesota, 1994.

[16] Allen, J.F., "Maintaining Knowledge about temporal intervals," *Communications of the ACM*, 1983. 26(11).

[17] Fujikawa, K., et al. "Multimedia Presentation System Harmony with Temporal and Active Media," in *USENIX*. 1991. Nashville, TN.

[18] Horn, F. and J.B. Stefani, "On Programming and Supporting Multimedia Object Synchronization," *The Computer Journal*, 1993. 36(1): p. 4-18.

[19] Esch, J.W. and T.E. Nagle. "Representing Temporal Intervals Using Conceptual Graphs," in *Proc. 5th Annual Workshop on Conceptual Structures.* 1990.

[20] Buchanan, C.M. and P.T. Zellweger. "Scheduling Multimedia Documents Using Temporal Constraints," in *Network and Operating Systems Support for Digital Audio and Video. Third Int'l Workshop Proceedings.* 1992. Germany.

[21] Gibbs, S. "Composite Multimedia and active objects," in *OOPSLA*. 1991.

[22] Pazandak, P. and J. Srivastava, "A Multimedia Temporal Specification Model and Language," Technical Report 94-33, University of Minnesota, 1994.

[23] Blair, G., et. al., "Formal Support for the Specification and Construction of Distributed Multimedia Systems," School of Engineering, Computing and Mathematical Sciences, Lancaster University, England. Internal Report MPG-93-23, 1993.

[24] Schnepf, J., J. Konstan, and D. Du, "Doing FLIPS: FLexible Interactive Presentation Synchronization," Technical Report 94-49. University of Minnesota, 1994.

[25] Hudson, S. and C. Hsi, "The Walk-Through Approach To Authoring Multimedia Documents," in *ACM Multimedia 94*. 1994. San Francisco, California.

[26] Vander, A., J. Sherman, D. Luciano, *Human Physiology: The Mechanisms of Body Function*, 3rd edition, McGraw-Hill, 1980.

[27] Paxson, V., C. Saltmarsh, "Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems," *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, January, 1993.

[28] Pazandak, P., J. Srivastava, "The Language Components of DAMSEL: An Embedable Event-driven Declarative Multimedia Specification Language" *SPIE: Electronic Imaging International Conference, Photonics East '95*, Pittsburg, PA., 1995.