Streaming and Synchronization of Re-executable Content

Nelson R. Manohar
IBM Thomas J. Watson Research Center
Yorktown Heights, NY USA 10598
nelsonr@watson.ibm.com

ABSTRACT

This paper presents a framework and associated modeling and representation for the streaming and playback of re-executable content on heterogeneous network computing appliances.

1. INTRODUCTION

A computer-based workspace consists of multiple tools/applets. A computer-based workspace may span one or more applets and their appliances. One could readily imagine a workspace composed of TV content augmented by computations such a wire-mesh models, etc. Because of the size of these workspaces, the streaming of a workspace would be of interest in the provisioning of heterogeneous content (e.g., web-televisions and web-telepresence). In particular, we consider the notion of a "*replayable workspace*" (Fig. 2) to allow capture, storage, and playback of a session on the collective workspace spanned by multiple applets. During capture, the state of, and inputs to, applets in a workspace are recorded as temporal media streams in an intermediary representation, referred throughout this paper as $\psi()$ — a platform-independent re-executable content stream composed of time-ordered inputs to, and state of, an applet.

Consider the workspace, shown in Fig. 1, composed of four applets: (A) a mathematical notebook tool, (B) a graphing display tool, (C) an audio-annotation tool, and (D) a pointer-gesturing tool. Each applet operates over a media stream $\psi()$ consisting of time-ordered inputs and state. For example, $\psi(A)$ consists of `Insert()` and `Delete()` operations over the notebook; $\psi(B)$ consists of `Display()` and `Refresh()` of time series; $\psi(C)$ consists of `Play()` and `Record()` of audio frames; and $\psi(D)$ consists of operations such as `Get()` and `Dispatch()` of pointer movements. The use of a re-executable content representation makes possible the reuse and manipulation of the session and its underlying workspace at different times. The use of an intermediary representation makes possible the above on heterogeneous client workstations. However, the playback of such session requires temporal and causal orchestration of the re-execution $\psi()$-events. Through this paper, we assume the existence of: (1) a network-centric session management model (e.g., network computing, mobile computing, etc.); (2) some heterogeneous clients α and β (e.g., laptops, palmtops, etc.); (3) some unknown intermediary streaming representation $\psi()$ between these clients and the session manager; and (4) a network-centric state checkpointing mechanism (A/B DB). These components are shown in Fig. 3.

In particular, we consider the **spatial** and **temporal** reuse of our replayable workspaces (see Fig. 4). Spatial reuse occurs when W_β is different from W_α. As opposed to process migration, spatial reuse of a workspace allows the recreation and evolution of a workspace from the last known/safe workspace checkpoint ($W_{t(i)}$) stored in the network. Temporal reuse occurs when W_β is the same as W_α — *but at a later time*. As opposed to workspace checkpointing, temporal reuse of a workspace allows the recreation and evolution of a workspace from the last known/safe workspace checkpoint ($W_{t(i)}$) in the network. One benefit of temporal reuse is that it relaxes both the spacing between workspace checkpoints ($|t_i - t_{i-1}|$ as well as their storage requirements $|(W_{t(i)})|$ on the network-centric servers. Spatial-temporal reuse occurs when W_β is different from W_α — *and at a later time*. This service should be useful for nomadic/mobile users subject to unreliable/intermittent network access. The spatio-temporal reuse service can be used to support asynchronous collaborative work through iterative refinement of a session on a workspace ($W_{\alpha(0)}, W_{\alpha(1)}, W_{\alpha(2)}, \cdots$) by one or more users as we argued elsewhere.[12] Temporal reuse might prove useful for browsing other people's work, too.

The playback of these re-executable content sessions requires the streaming of $\psi()$-streams from a remote network repository to one or more client workstations. The re-execution of these

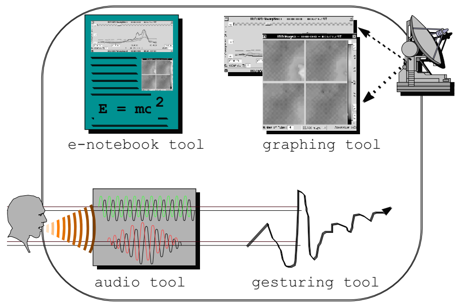

Figure 1. A workspace consists of multiple tools. Its playback requires temporal orchestration of the individual re-execution of its tools.

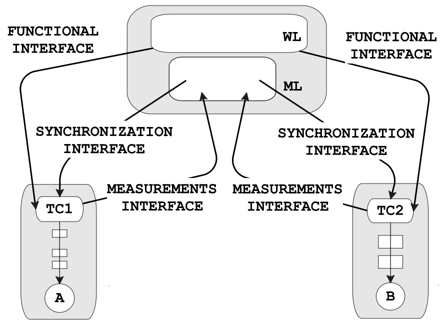

Figure 2. A session manager uses the synchronization interface to enforce media-independent relationships between multiple $\psi()$-streams. Applets provide periodical feedback reports about the efficiency of session management v

$\psi()$ streams recreates the workspace spanned by these applets on a receiving client workstation. However, because client workstations are heterogeneous, mechanisms are needed to ensure integrated re-execution of multiple $\psi()$-streams. Unlike the streaming of continuous multimedia (CM), the $\psi()$-streams are stateful, asynchronous, and aperiodical. These heterogeneous characteristics reduce the feasibility of existing continuous media integration solutions, in particular, when the heterogeneity of client workstations is considered. To playback such workspaces, we need to address the impact of asynchronous re-executable content over media integration.

The re-execution problem can be reduced to a temporal ordering problem. During capture of a session, since applets are not distributed across workstations, $\psi()$-streams share a common logical time system (LTS). Thus, during capture of a session, a full temporal order exists between events across different $\psi()$-streams. Because re-execution of $\psi()$ events is asynchronous and likely to occur on heterogeneous client workstations, the playout time of $\psi()$ events is unpredictable. Consequently, during playback of a session, only a partial temporal order exists between applet events across multiple $\psi()$-streams. Although the playback problem can be formulated as enforcing a full temporal order between partially-ordered streams, the satisfaction of this goal is constrained by the individual requirements of applets such as need for causality event ordering and/or sensitivity to jitter or variations over its continuity. Furthermore, differences in processor performance introduce asynchrony trends during the playback of re-executable content streams on client workstations of unpar performance with respect to the recording site.

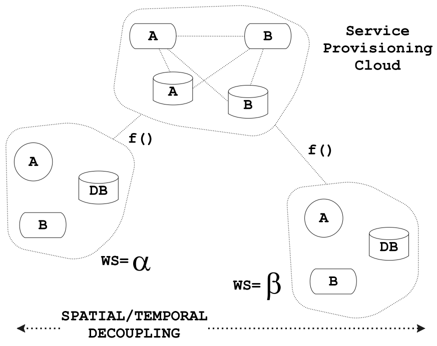

Figure 3. We assume the existence of a network-centric session manager; heterogeneous workstation clients α and β that subscribe to our replay-awareness service provisioning; an intermediary streaming representation $\psi()$ between these clients and the session manager; and a state checkpointing mechanism (A/B/DB).

This paper presents a framework for remote integration of re-executable content requiring syncronization to continuous media for playback on heterogeneous platforms. This paper is structured as follows. First, we review related trends. Second, we present our framework for remote playback coordination. Last, we present our conclusions.

2. RELATED TRENDS

The transportable workspaces discussed in this paper are different from the notion of teleporting[15] which addresses the mobility of the user interface to a workspace (i.e., its "*look-and-feel*"). Instead, we address the mobility and evolution of a computation and its user interface. The mobility of a computation has been addressed by research on process migration for homogeneous environments. We rely on the capture and replay of an intermediary representation between the session manager and its network clients. Research on intermediary execution representation as a mechanism to support dynamic process mobility is undergoing, for example elsewhere.[8] Java provides platform-unaware content mobility through the re-execution of downloadable content applets and workspaces.

We reused ideas found in the interactive playback of stored continuous media. To a well-versed reader in multimedia, our framework could be accurately specified as media-independent policies for the playback of relative interval-based scheduling of fine-grained asynchronous events subject to relative timing constraints (both causal and temporal) on multiple n-ary relationships that are enforced through pair-wise (binary) synchronization points. To preserve temporal correspondence on the playback of asynchronous media, we rely on relative scheduling. To avoid frequent synchronization overheads, we rely on enforcement of temporal intervals through point synchronization. The specification of asynchronous n-ary relationships between temporal intervals has been modeled through timed petrinets, most markedly, in the object composition petrinets (`OCPN`)[11] and time-flow graphs (`TFG`).[10] Asynchronous intermissions during playback are modeled in `TFG` through intermission nodes that buffer the asynchrony between concurrent relative temporal intervals. Gap filtering is used to remove playout gaps; by recomputing a run-time schedule for the media playout. However, scheduling in `TFG`s lacks the notion of removal of long-term playback trends (which occur when scenarios are played back on workstations of different performance to the record workstation). It is arguable whether zero-gap playback (a short-term optimization) achieves playback smoothness (a long-term requirement).

Many systems enforce full temporal order of asynchronous events with tight synchronization but without continuity guarantees. Examples of these include schedulers used in distributed event simulations[1,6,13] and causality preserving protocols such as Birman's `ISIS`.[2] Our research problem can be formulated as enforcing playback continuity over an asynchronous event simulation. Several systems enforce full temporal order of asynchronous events with synchronization and continuity but over coarse event grain. These include schedulers in multimedia authoring systems.[4] Abstractly, the main difference comes from coarseness of the events they handle. These schedulers deal with high level asynchronous events, more correctly referred to as media parts (such programs and transitions). The coarseness of this grain allows the support of complex temporal relationships. On the other hand, we focus on fine-grained asynchronous events. These events are typically found inside a single media part. Both approaches are needed and complementary.

There are systems for the enforcement of full temporal order of fine-grained synchronous or periodical tasks with tight synchronization and strong

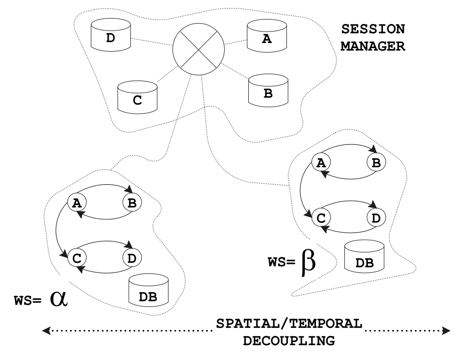

Figure 4. Spatial and temporal reuse of transportable workspace sessions.

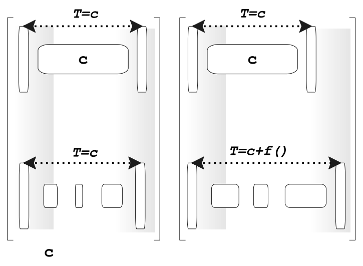

Figure 5. The playback for $\psi()$ events is asynchronous and has unpredictable playout duration (top), much unlike continuous media (bottom). Unpredictable playout time introduces departures between record (left) and playback timing (right).

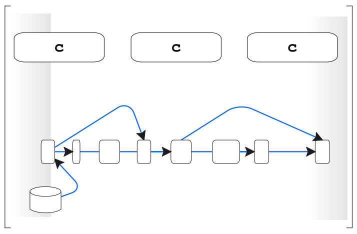

Figure 6. Unlike with continuous media (top), $\psi()$-streams (bottom) are stateful. Statefulness affects playback mechanisms. Playback (either forward or backwards) from an $\psi()$ event must account for stateful dependencies.

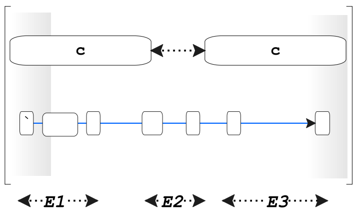

Figure 7. Continuous media (top) is divided into frames of playout duration c. Its playback characterization for continuous playback is prescribed by c. Periodicity is a typical assumption in the integration of continuous media. On the other hand, an $\psi()$-stream (bottom) lacks periodicity nor characterization.

playback continuity. Some examples are schedulers used for playback of continuous media in general purpose systems[11] and on real-time systems.[17] However, these scheduling algorithms benefit (on one degree or another) from properties of continuous media not found in $\psi()$-streams. For example, continuous streams have well-defined events with predictable statefulness (if any); neither dropping nor inserting arbitrary events is possible during the streaming of re-executable content (see Fig. 6). Similarly, scheduling mechanisms for continuous streams[5] often rely on the fact that continuous media events (i.e., media frames) have predictable playout times (see Fig. 5). On the other hand, because of the asynchronous nature of re-executable content, event playout time is unpredictable. Finally, continuous media is divided into frames of expected playout time c (see Figs. 7). The playback of continuous media can be prescribed by c since during processing and scheduling, c provides an *a-priori, constant* bound. For example, consider the following constraints: (1) display a frame every $c = 33ms$, (2) present each frame for a maximum duration of $c = 33ms$, (3) control discontinuities between frames to less than $10ms$. Such a-priori knowledge about the periodicity of event playback is an inherent assumption of many mechanisms for the playback of continuous media.[16] This is particularly true of deadline-based scheduling of continuous media.[5] On the other hand, a re-executable content stream lacks periodicity.

3. FRAMEWORK

A workspace W of n heterogeneous streams is specified by a tolerance matrix W_T, which provides a compact characterization of all the relationships between streams in the workspace as follows:

$$W_T = \left\{ \begin{array}{cccccc} -1 & d_{12} & \cdots & d_{1j} & \cdots & d_{1n} \\ d_{21} & -1 & \cdots & d_{2j} & \cdots & d_{2n} \\ \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ d_{i1} & d_{i2} & \cdots & d_{ij} & \cdots & d_{in} \\ \cdots & \cdots & \cdots & \cdots & \ddots & \cdots \\ d_{n1} & d_{n2} & \cdots & d_{nj} & \cdots & -1 \end{array} \right\} \quad (1)$$

where d_{ij} is a short-hand for $d(i \rightarrow j)$ — that is, a_i releases a_j when a_j is ahead. This corresponds to the synchronization relationship $R(a_i \leftarrow a_j)$. Each entry d_{ij} of W_T characterizes the asymmetrical relationship between i and j in terms of a "*meet*"

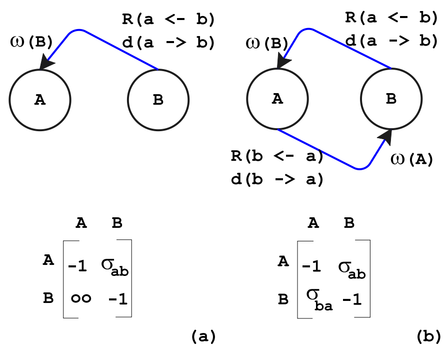

Figure 8. 1-way (left) and 2-way relationships between two streams and associated W_T matrices.

constraint as follows:

$$d_{ij} = \left\{ \begin{array}{lcl} \sigma_{ij} & : & for\ i \neq j; \\ -1 & : & for\ i = j; \end{array} \right\} \qquad (2)$$

A "*meet*" constraint $d_{ij} = \sigma_{ij}$ constrains the playback of two streams i and j in terms of: (1) a dependency constraint d_{ij} and (2) the tolerance σ_{ij} of the dependency constraint d_{ij} to the estimated asynchrony referred to as ϵ^*_{ij}. Note that the diagonal elements d_{ii} are set to -1. This justs flags the fact that no "*meet*" constraint exists between a stream and itself. In all other cases, the "*meet*" constraint is specified through a single parameter σ_{ij}. This parameter is referred to as the "*tolerance of j to asynchrony from i*".

This scheme allows flexible enforcement of "*meet*" constraints that allows us to address heterogeneity on synchronization relationships. Our three treatment policies (absolute, lazy, and laissez-faire) can be expressed by this framework by characterizing the corresponding σ_{ij} values as follows:

$$\sigma_{ij} = \left\{ \begin{array}{lcl} 0 & : & absolute \\ 0 < c < \infty & : & lazy \\ \infty & : & laissezfaire \end{array} \right\} \qquad (3)$$

The tolerance matrix W_T captures the asymmetry of 1-way relationships as the coupling of an unconstrained *laissez-faire* and a dependency constraint. This asymmetry is observed in transposable entries as:

$$1 - way : \left\{ \begin{array}{l} d_{ij} = \infty \\ 0 \leq d_{ji} < \infty \end{array} \right\} \qquad (4)$$

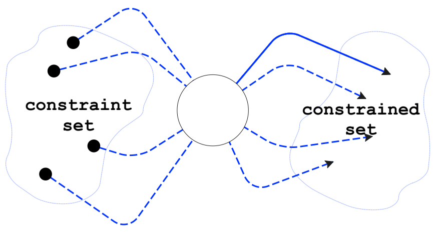

Figure 9. The view of the constraint handling problem as seen by an arbitrary stream j. The stream j sees the workspace as divided into two sets: (1) constraint-set — the set of streams that impose a "*meet*" constraint over the release of j, that is, i *such that* $(0 \leq d_{ij} < \infty)$. (2) constrained-set — the set of streams over which j imposes a "*meet*" constraint, that is, k *such that* $(0 \leq d_{jk} < \infty)$. These sets are not mutually exclusive.
.

The tolerance matrix W_T also captures the symmetry of 2-way relationships as the coupling of two identical dependency constraints. This symmetry is observed in transposable entries as follows.

$$2 - way : \left\{ \begin{array}{l} 0 \leq d_{ij} < \infty \\ 0 \leq d_{ji} < \infty \end{array} \right\} \ \& \ d_{ij} = d_{ji} \qquad (5)$$

Fig. 8 illustrates the modeling difference between 1-way and 2-way relationships with respect to two streams a and b. Again, consider the workspace in Fig. 10. Such workspace can be specified by the following tolerance matrix $W_T(A, B, C, D) =$

$$\left\{ \begin{array}{cccc} -1 & \sigma_{ab} = 0 & \infty & \infty \\ \sigma_{ba} = 0 & -1 & \infty & \infty \\ \sigma_{ca} = k_1 & \infty & -1 & \sigma_{cd} = k_2 \\ \infty & \infty & \sigma_{dc} = k_2 & -1 \end{array} \right\} \qquad (6)$$

where k_1 is a tolerance specification. The tolerance matrix W_T provides a precise and concise representation of the workspace specification $W = \{applets, dependencies, tolerances\}$ as well as it provides a foundation for the analysis of its media integration properties.

Finally, the transpose of the tolerance matrix is the **constraint matrix** (W_C). Equivalently,

$$W_C = W_T^t = \left\{ \begin{array}{cccccc} -1 & c_{12} & \cdots & c_{1j} & \cdots & c_{1n} \\ c_{21} & -1 & \cdots & c_{2j} & \cdots & c_{2n} \\ \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ c_{i1} & c_{i2} & \cdots & c_{ij} & \cdots & c_{in} \\ \cdots & \cdots & \cdots & \cdots & \ddots & \cdots \\ c_{n1} & c_{n2} & \cdots & c_{nj} & \cdots & -1 \end{array} \right\} \tag{7}$$

where $c_{ij} = d_{ji}$; a short-hand for $d(j \to i)$.

The playback of W (see Fig. 10) requires the preservation of three fine-grained synchronization relationships between applet streams. Relationship R1 is a 2-way causal relationship since A and B are discrete streams with strong temporal correspondence. The goal of such relationship is to achieve smooth playback of both streams subject to the constraint of preserving causality between events across streams. Causality here is bi-directional across events from both streams (A to B) and (B to A). Relationship R2 is a 1-way temporal relationship since A is discrete and C is continuous with relaxed temporal mutual correspondence. Its goal is to achieve smooth playback of A and playback continuity on C while preserving the relative timing during the streaming between inputs from A and C. This relationship is not causal. Relationship R3 is a 2-way temporal relationship since C is continuous and D is discrete with relaxed temporal correspondence. Its goal is to achieve playback continuity on C and smooth playback of D while preserving relative timing during the streaming between inputs from C and D. This relationship is not causal. Relationship R1 arises from the need to support causal dependencies such as "cut and paste" between notebook and graphing tools. R2 arises from the need to preserve temporal correspondence of audio-annotations over the playback of the notebook. Finally, R3 arises from the need to preserve temporal correspondence of audio-annotated pointer-gesturing.

A cautious reader should have note that, under apparently identical circumstances, R2 is 1-way relationship while R3 is a 2-way relationship. This is a design decision; the sort of decision typically made by the designer of a workspace. Let's analyze such decision. Throughout this discussion, recall that a tight synchronization pair is a 2-way set between two streams. During playback, whenever the 2-way set $(A : B)$ is ahead of the 2-way set $(C : D)$, the 2-way set $(A : B)$ attempts to "*meet*" with C and in this case, with the 2-way set $(C : D)$ — a result of its 1-way relationship $R(C \leftarrow A)$. In such cases, the 2-way set $(A : B)$ is forced to wait for the $(C : D)$ set. Here, the 1-way relationship favors continuity on the playback of the 2-way set $(C : D)$ to continuity on the 2-way set $(A : B)$ while it preserves tight synchronization between the two 2-way sets under this condition. Fig. 11 illustrates this condition; playback continuity on the 2-way set $(C : D)$ is unaffected during time $[t_1 \cdots t_2]$ by what happens with the 2-way set $(A : B)$. However, the 2-way set $(A : B)$ is forced to wait for the 2-way set $(C : D)$ during time $(t_2 - t_1)$. Synchronization between the two 2-way sets is achieved at time t_2 at the expense of a discontinuity $(t_2 - t_1)$ on the 2-way set $(A : B)$. If an underlying asynchrony exists between the two 2-way sets, a long-term scheduling compensation over one of the 2-way sets might be desirable. On the other hand, whenever the 2-way set $(A : B)$ is behind the 2-way set $(C : D)$, the 2-way set $(A : B)$ continues — a result of the asymmetry of the 1-way relationship $R(C \leftarrow A)$. Again, this decision favors continuity on the playback of the 2-way set $(C : D)$ by relaxing the synchronization between these two 2-way sets. Fig. 12 illustrates this condition; playback of the 2-way set $(A : B)$ is not disrupted by C at time t_1. Again, playback continuity of the 2-way

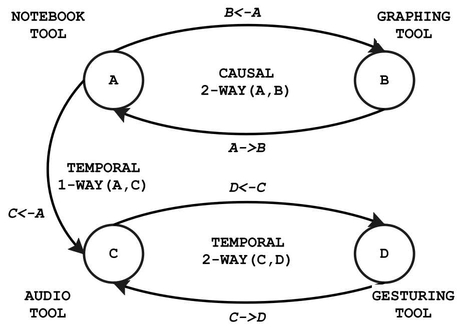

Figure 10. Relationships between events of different applets may be different. Playback of the workspace requires preserving these relationships. For example, the playback of the notebook and graphing tools requires enforcing a causal relationship (such as cut and paste) between them. On the other hand, the playback of audio-annotated pointer gesturing requires only a temporal (non-causal) relationship.

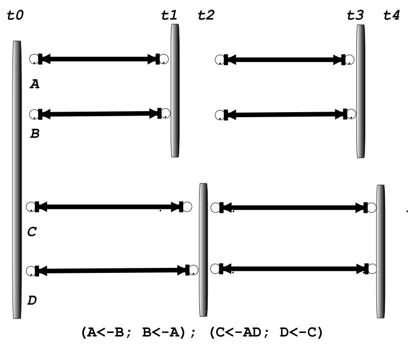

Figure 11. Here, the A:B pair re-executes faster than the $C : D$ pair. Each pair is kept synchronized by independent 2-way relationships $R_1(A : B)$ and $R_3(C : D)$. However, the 1-way relationship $R_2(C \leftarrow A)$ forces the $A : B$ pair to synchronize to the $C : D$ pair. Playback continuity on the $C : D$ pair is unaffected while the $A : B$ pair observes a discontinuity of duration $t_2 - t_1$.

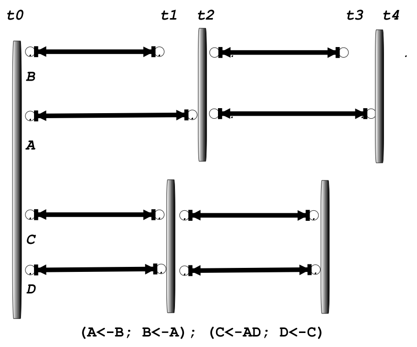

Figure 12. Here, notebook (A) and graphing (B) tools re-execute slower than audio (C) and gesturing (D) tools. While 2-way relationships preserve synchronization on $A : B$ and $C : D$, the 1-way relationship $R_2(C \leftarrow A)$ allows a relaxation of the temporal correspondence between $A : B$ and $C : D$ by $(t_2 - t_1)$. Synchronization between (*relationship*) pairs is traded-off for continuity on the $C : D$ pair.

set $(C : D)$ remains unaffected. Playback continuity of the 2-way set $(A : B)$ is also unaffected. Synchronization between the 2-way sets $(A : B)$ and $(C : D)$ is relaxed by time $(t_2 - t_1)$. Finally, the temporal correspondence inside each 2-way set is kept tight. That is, synchronization between the two 2-way sets is traded-off for continuity on each 2-way set. If an underlying asynchrony exists between the two 2-way sets, a long-term scheduling compensation over one of the 2-way sets may be necessary.

The session manager handles integration of its n heterogeneous media streams through periodical point-based constraint resolution. The workspace appears to the session manager as being temporally-based but in a rather chaotic state in terms of constraint management. Fig. 13 illustrates the sort of temporal chaos perceived by the session manager during an arbitrary scheduling interval of n streams. Fig. 9 shows how an arbitrary stream j perceives its relationships to other streams on the workspace. Stream j perceives the workspace to be divided in two sets: a constraint-set and a constrained-set. The constraint-set represents as the set of all streams that impose a "*meet*" constraint over the release of j. The constrained-set represents the set of all streams over which j imposes a "*meet*" constraint. To applets, the workspace appears constraint-based and atemporal.

Next, we argue the correctness of our heteroge-

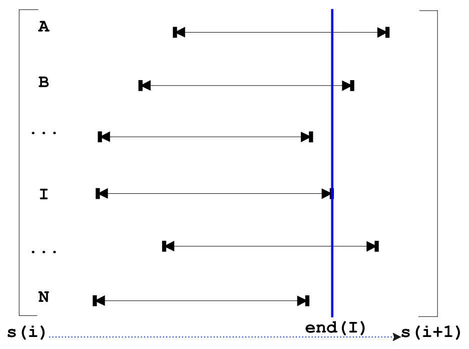

Figure 13. Temporal view to constraint handling on an arbitrary scheduling interval across n streams $(A \cdots I \cdots N)$. The lenght of the i^{th} horizontal line segment represents the relative playout duration of the scheduling interval on the i^{th} stream. Playout duration is unpredictable for re-executable content.

neous media integration for an arbitrary stream j with respect to both its constraint and its constrained sets. First, let's sort the temporal view of an arbitrary scheduling interval by increasing "finish" times for each of the n individual scheduling intervals. The result should be similar to that shown in Fig. 14.

Media integration relies on "*meet*" constraints that are evaluated for every synchronization dependency in the constraint set of j (see Fig. 15). The release of j is constrained by each "*meet*" con-

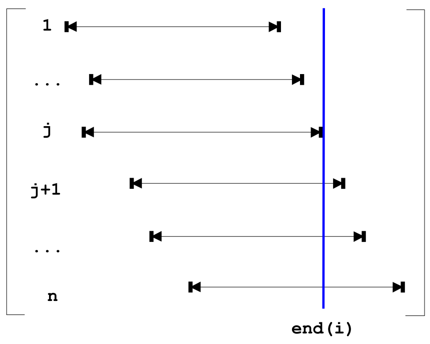

Figure 14. Temporal sort of the scheduling interval by increasing "finish" times.

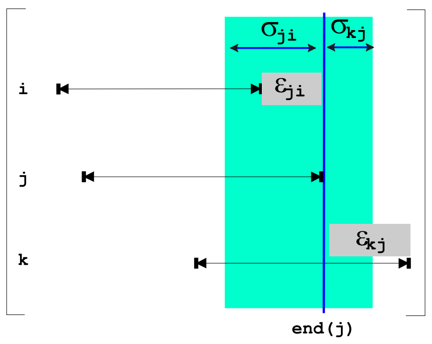

Figure 15. The release of j is constrained by all the "*meet*" constraints on its constraint set. However, the parameter σ_{ij} relaxes a "*meet*" constraint c_{ji} by tolerating some asynchrony between streams j to i. During run-time, such asynchrony ϵ^*_{ji} is estimated. A "*meet*" constraint is satisfied when $-\sigma_{ij} < \epsilon^*_{ji}$.

straint in this set. However, the parameter σ_{ij} relaxes the "*meet*" constraint of c_{ji} by tolerating some asynchrony between j and i. During run-time, a smoothed indicator of the asynchrony between these two streams ϵ^*_{ji} is computed. Note that ϵ^*_{ij} is negative when j ahead of i. A "*meet*" constraint is satisfied when $-\sigma_{ij} < \epsilon^*_{ji}$. A "*meet*" constraint $c_{ji}(a_j \rightarrow a_i) = \sigma(a_i, a_j)$ between streams a_j and a_i is satisfied when the asynchrony indicator ϵ^*_{ij} is estimated to be within statistically tolerable limits (say σ_{ij}). Equivalently,

$$meet(a_i, a_j) = \begin{Bmatrix} YES & -\sigma_{ij} < \epsilon^*_{ji} \\ NO & otherwise \end{Bmatrix} \qquad (8)$$

The constrained (OUT) set of a stream a_i is composed of nodes a_k connected by a direct outgoing edge from a_i in W_C — note that it says W_C not W_T. Equivalently,

$$OUT(a_i) = a_k \text{ such that } (0 \leq d_{ik} < \infty) \qquad (9)$$

Synchronization-wise, a stream $a_k \in OUT(a_i)$ has a dependency constraint d_{ik} with a_i. At every scheduling interval, when a_k is ahead of a_i, a_k "*meets*" a_i before continuing. Scheduling-wise, a_k adapts its playback to a_i. For example, consider the workspace specification on Equation 7.12 where $OUT(A) = \{B\}$; $OUT(B) = \{A\}$; $OUT(C) = \{A, D\}$; and $OUT(D) = \{C\}$.

The constraint set (IN) of a stream a_i is composed of all incoming edges from some a_j to a_i in W_C. Equivalently,

$$IN(a_i) = a_j \text{ such that } (0 \leq d_{ji} < \infty) \qquad (10)$$

Synchronization-wise, a stream $a_j \in IN(a_i)$ imposes a dependency constraint d_{ji} over a_i — respectively, $R(a_j \leftarrow a_i)$ holds. At every scheduling interval, when a_i is ahead of a_j, a_i must "*meet*" a_j before continuing. Scheduling-wise, a_i adapts its playback to the $IN(a_i)$. For example, consider the workspace specification on Equation 7.12 where $IN(A) = \{BC\}$; $IN(B) = \{A\}$; $IN(C) = \{D\}$; and $IN(D) = \{C\}$.

The outlier set $IN^*(a_i)$ is the *run-time subset* of streams in $IN(a_i)$ for which the asynchrony indicator $\epsilon^*(a_j, a_i)$ exceeds its tolerance $\sigma(a_j, a_i)$. Equivalently, $IN^*(a_i) =$

$$a_j \in \{IN(a_i) \text{ such that}(|\epsilon^*(a_j, a_i)| > \sigma(a_j, a_i))\} \qquad (11)$$

The outlier set of $OUT^*(a_i)$ is the *run-time* subset of streams in $OUT(a_i)$ for which the asynchrony indicator $\epsilon^*(a_i, a_k)$ exceeds its target tolerance $\sigma(a_i, a_k)$). Equivalently, $OUT^*(a_i) =$

$$a_k \in \{OUT(a_i) \text{ such that}(|\epsilon^*(a_i, a_k)| > \sigma(a_i, a_k))\} \qquad (12)$$

A run-time outlier subset of a stream a_i represents any stream a_j having either a dependency constraint d_{ij} (for $OUT(a_i)$) or d_{ji} (for $IN(a_i)$) with stream a_i for which (*during run-time*) a statistically significant asynchrony departure is observed (see Fig. 16).

A negative outlier set $S^-(a_i)$ is the subset of streams in an outlier set $S^*(a_i)$ for which the

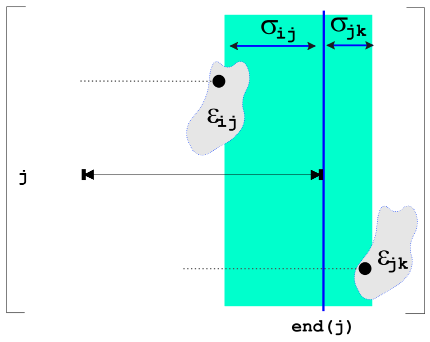

Figure 16. The coupling of the temporal ordering and the "*meet*" constraint during the handling of multiple 1-way temporal relationships. Two disjoint sets are created during run-time over the temporal view of a scheduling interval: the set of streams that completed significantly before j (positive outliers for an ϵ_{ij}); and the set of streams that complete significantly after j (negative outliers for an ϵ_{kj}).

asynchrony indicator is negative. Equivalently, $S^-(a_i) =$

$$a_j \in \{S^*(a_i) \text{ such that } (\epsilon^*(a_i, a_j) \leq -\sigma(a_i, a_j)\} \quad (13)$$

A positive outlier set $S^-(a_i)$ is the subset of streams in an outlier set $S^*(a_i)$ for which the asynchrony indicator is positive. Equivalently, $S^+(a_i) =$

$$a_j \in \{S^*(a_i) \text{ such that } (\epsilon^*(a_i, a_j) > \sigma(a_i, a_j))\} \quad (14)$$

We define a run-time scheduling conflict over a stream j as a temporal view applied over a tolerance matrix for which distinct statistically significant trends exist. There are two cases to consider. First, a magnitude conflict occurs when an outlier set of $OUT^*(a_j)$ is not empty and contain two or more distinct (statistical significant) trends. Second, a sign conflict occurs when both positive and negative outliers of $OUT^*(a_j)$ are not empty and each contains one or more (statistical significant) trends.

Example 1: Let a, b, c be streams for which the synchronization relationships $R(b \leftarrow a)$ & $R(c \leftarrow a)$ hold. A run-time (sign-based) scheduling conflict for a arises in any scheduling interval whenever the following condition is met.

$$|\epsilon^*_{ba}| > \sigma_{ba} \ \& \ |\epsilon^*_{ca}| > \sigma_{ca} \ \& \ \frac{\epsilon^*_{ba}}{\epsilon^*_{ca}} < 0 \quad (15)$$

The conflict arises because a requires a positive scheduling compensation over $R(b \leftarrow a)$ while a negative scheduling compensation is required over $R(c \leftarrow a)$.

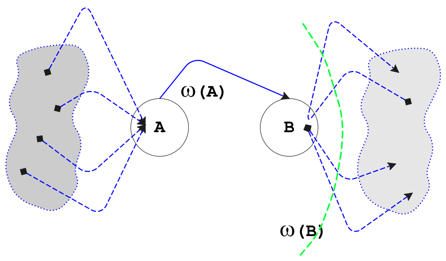

Figure 17. A synchronization conflict is possible when $\|IN()\| > 1$ (see A). A scheduling conflict is possible when $\|OUT()\| > 1$ (see B).

Scheduling conflicts produce at worst $r = \|IN^*(i)\|$ distinct pair-wise schedule compensations $\delta_{ji} = f(\epsilon_{ji}, \sigma_{ji})$ during run-time. In general, there are three possible ways to combine such pair-wise schedule compensations into one global schedule compensation $\hat{\delta}_j$.

- $\hat{\delta}_i = \delta_{ji}$; — for *a given* j.
- $\hat{\delta}_i = f(\delta_{ji})$; — for *some* smoother $f()$.
- $\hat{\delta}_i = 0$; — i.e., select none.

It is unclear whether choosing the largest, smallest, k^{th} largest, mean, median, etc. produces a globally optimal compensation — or even whether one such exist. Across scheduling intervals, the first two policies could easily over- as well as under-compensate streams in $IN^*(i)$. This limits our ability to handle, with predictable behavior, arbitrary n-ary dependencies to only those cases where $r \leq 1$; that is, when the $IN(i)$ has only one temporal relationship. However, through the coupling of scheduling with synchronization measures associated with temporal 2-way relationships it is possible to extend predictable behavior to the handling of a special case of n-ary dependencies.

Example 2: Let a, b, c, d be streams for which the synchronization relationships $R(a : b)$ & $R(c : d)$ holds. Any tight synchronization pair $(i : j)$ (any 2-way relationship) is by definition a strongly connected component (SCC) — as dependencies edges leave and come from either of its two nodes. For example, streams a & b form $SCC(a, b)$. Similarly, c & d form $SCC(c, d)$. SCCs guarantee tight synchronization with respect to all its members. However, these SCCs are unrelated. Consider now, adding a temporal 1-way relationship $R(c \leftarrow a)$; that is, a adapts to c, implemented through a dependency constraint $d_{ac}(a \rightarrow c)$. This new edge has the effect of slaving the $SCC(a, b)$ to the $SCC(c, d)$ (as shown in Figs. 11 and 12).

Example 3: Let a, b, c, d be streams for which the synchronization relationships $R(a : b)$ & $R(c : d)$ holds. Consider adding a 2-way synchronization relationship $R(a : c)$. This has the effect of merging the two original SCCs into a single one: $SCC(a, b, c, d) = \{SCC(a, b) + SCC(c, d)\}$. Such SCC guarantees tight synchronization with respect to all the streams.

The two previous examples illustrate two fundamental ideas. First, SCCs provide the basis for tight synchronization (a predictable behavior by definition) and second, a temporal 1-way synchronization relationships can be used to couple two SCCs in such a way so as to achieve predictable behavior. For any stream on a given SCC, a scheduling measure might be desirable but it is not required (as synchronization measures guarantee tight synchronization). This observation allows us to achieve predictable behavior during the handling of the following n-ary dependency constraints. Predictable behavior is possible for an $IN(j)$ set for which exactly $||IN(j)|| - 1$ streams have a 2-way relationship with j. Such streams form a SCC with j. Scheduling conflicts can never arise (since the schedule compensation of j would be determined by only one stream). For example, consider node A on Fig. 10 — the $IN(A) = \{B, C\}$, where B has a 2-way relationship with A (that is $(A : B)$). Under this setup, only A adapts to C.

An SCC can adapt its playback, too. The key problem for adaptable SCCs is a problem referred to as the convergence of its members. Scheduling conflicts arise when a stream attempts to enforce more than two or more temporal 1-way relationships. On the other hand, the convergence problem arises when two streams form an adaptable tight synchronization pair (an SCC) of exactly two temporal 1-way synchronization relationships causing each stream to adapt to the other. Both problems, however, share similar solutions. The approach is to preempt the existence of the problem. For example, we can preempt the convergence problem by allowing only one member of an adaptable tight synchronization pair to adapt. Equivalently, for any stream j in a SCC, we allow only one edge in $IN(j)$ to be related to a temporal 1-way relationship. As a result, for any stream in a SCC, no stream adapts to more than one neighbor so the convergence problem is avoided. We define a run-time synchronization conflict over a stream j as a temporal view over an associated tolerance matrix that exhibits statistically significant trends for some streams to fall behind j and, for others, to fall ahead of j. It follows that a run-time synchronization conflict on a_i occurs only when both positive and negative outliers of $IN^*(a_j)$ are not empty.

Example 4: Let a, b, c be streams for which the synchronization relationship $(a \leftarrow bc)$ holds. A run-time synchronization conflict for a arises in any scheduling interval whenever the condition is met.

$$|\epsilon^*_{ba}| > \sigma_{ba} \ \& \ |\epsilon^*_{ca}| > \sigma_{ca} \ \& \ \frac{\epsilon^*_{ba}}{\epsilon^*_{ca}} < 0 \qquad (16)$$

Such conflict arises because a should wait for b to catch-up while on the other hand, it should continue too, so as to keep up with c.

Synchronization conflicts do not represent a problem. Stream a_j simply waits for streams in its positive outlier $IN^+(a_j)$ so as to satisfy its pairwise "*meet*" constraints while on the other hand, because of the asymmetry of the "*meet*" constraint, no action is needed for streams in the negative outlier $IN^-(a_j)$.

3.1. The $MediaLayer()$ Algorithm

Tables 1 through 4 contain the specification of the media integration algorithm for n-ary dependency constraints — when restricted as discussed in the

```
$MediaLayer()\{$
    $OK = WorkspaceLayer();$

    $while(OK)\{$
        $j = GetNextUpdate();$
        $UpdateConstraintsWith(j);$
        $HandleUpdateFor(j);$
        $OK = WorkspaceLayer();$
    $\}$
$\}$
```

Table 1. The *MediaLayer*() Algorithm.

```
$HandleUpdateFor(j)\{$
    $MeetWithSet(j, IN^*(j));$
    $AdaptToSet(j, IN^*(j));$
    $ReleaseStream(j);$
$\}$
```

Table 2. The *HandleUpdateFor*() Algorithm.

```
$MeetWithSet(j, IN^*(j))\{$
    $\forall_i \in IN^-(j)$
        $meet(i, j);$
$\}$
```

Table 3. The *MeetWithSet*() Algorithm.

```
$AdaptToSet(j, IN^*(j))\{$
    $if(\ \overline{\|IN^-(j)\|\&\|IN^+(j)\|}\ )$
        $adapt(max(|IN^*(j)|);$
$\}$
```

Table 4. The *AdaptToSet*() Algorithm.

previous section. The *MediaLayer* algorithm specified in Tables 1 performs four simple tasks: (1) wait for the next Update() from a stream j, (2) propagate the update to the global constraint resolution, (3) update the n-ary dependency resolution for j, and (4) couple with processing at the workspace layer. The *MediaLayer*() algorithm takes $O(n)$ time per update — since step 1 takes $O(1)$ time, step 2 takes $O(n)$ time, step 3 takes $O(n)$ time, and step 4 takes $O(1)$ time — for each of its n streams thus resulting in a total worst case time of $O(n^2)$.

The algorithm *HandleUpdateFor*() in Table 2 handles constraint resolution for the n-ary dependency constraint problem of stream j. After all the dependency constraints for stream j are met, a check is made to determine whether the schedule of stream j should be adapted or not. Then, stream j is released. The specification of the *MeetWithSet*() algorithm in Table 3. A stream j will not be released until all its dependency constraints are met since the *MeetWithSet*() algorithm returns only when all dependencies $d_{ij}(i \rightarrow j)$ are met. The proof follows from the definitions of the *meet*(i, j) constraint and from the definition of the negative outlier of $IN^*(j)$.

4. CONCLUDING REMARKS

Although the characterization of workspace seemed similar to that of multimedia authoring environments, we showed the temporal relationships found in replayable workspaces to be different — we showed them to be of fine-granularity and strongly heterogeneous. We characterized the playback of tools on a workspace in terms of multiple, co-existing, heterogeneous temporal relationships and then presented a temporal model for the specification and modeling of replayable workspaces in terms of these temporal relationships to then address their integration during playback. We showed these temporal relationships to be richly heterogeneous and formulated a media-independent specification of replayable workspaces so as to reduce the impact such heterogeneity. We showed the characterization of replayable workspaces in terms of heterogeneous temporal relationships and introduced a formal model for their integration.

This paper presented modeling arguments to support *mobility and reuse* of workspace content across heterogeneous network computing appliances. In the visions of future internet frameworks,[3,7,9] the notions of workspace checkpointing, process migration, and service/platform transparency to heterogeneous clients represent fundamental building blocks for the provisioning of next generation, network-centric, services. Throughout this paper, we assumed: (1) the existence of an intermediary representation $\psi()$ and (2) the ability to do state checkpointing. The streaming of re-executable content should arise whenever a remote repository is used to store state and inputs of one or more computer appliances, as for example in network computers and in next generation wireless (mobile/nomadic) computing appliances. As the quality and heterogeneity of these digital consumer appliances increases, the need for policies to integrate richer and more complex forms of interactions (i.e., re-executable content) will become more evident.

APPENDIX A. MODELING

Tool and media independence are central to our framework as mechanisms for the removal of heterogeneity between applets. A replayable applet is a small application, tool, applet, or device driver, that has been expanded to incorporate these mechanisms. A replayable applet is as unaware of other applets as the original applet was. Replayable applets are coordinated by a centralized process, referred to as the session manager. To replayable applets, media processing occurs *"behind the curtains"*. This applet-transparent media processing takes place at the session manager. The session manager also provides a unified user interface to the workspace spanned by its applets. Workspace management is enforced by the session manager, through operations such as { f_i = `record()`, `replay()`, `goto()`, `edit()`, `pause()`, `continue()`, $\cdots$ } that operate over one or more applets. The developer of a replayable applet is responsible for the implementation of these operations. Media processing is approached abstractly; the session manager is only aware of media-independent information such as (1) the number of applets on its workspace, (2) their inter-relationships, and (3) the tolerance of these relationships. This information is used in the characterization of the playback requirements of a workspace.

Tool and media are not totally decoupled. Workspace management is coupled to media processing as follows. Before initiating a new workspace management operation $f_i(...)$; the session manager waits for the stabilization of pending media processing, i.e., $f_{i-1}(), \cdots$. A computer-based workspace W is spanned by a set of tools or applets that provide integrated services to allow one or more users to perform a domain-specific task. This sort of workspace arises in domains such as space-science and computer-based radiography. Informally, a workspace W is described by three tuples: (1) W_A: the applets found in the workspace; (2) W_R: the relationships between these applets; and (3) W_T: the tolerances of these relationships. In our context, an applet A_i is a small and simple application such as a component-based tool, an active-pad control, a java applet, or even a device driver. Typically, applets have a single-purpose, well-defined, abstract base machine.

The notion of an abstract base machine was defined by Parnas.[14] Let $abm(A_i)$ be the abstract base machine of applet A_i, where $abm(A_i)$ is composed of abstract operations $(f_1(), f_2(), \cdots, f_n())$. Tool provide integrated services that spanned the workspace. A tool renders a user-oriented, well-defined, service to the later-time reuse of a workspace. Example of tool services are the record/replay of audio and pointer gesturing. A well-defined tool service $TS(A_i)$ is just a user-level representation of an underlying, well-defined, abstract base machine, i.e., $TS(A_i) = f(abm(A_i))$. Ideally, we desire to abstract $abm(A)$ so as to specify $TS(A_i) = abm(A_i)$. This allows us to specify semantical queries over the usage of an applet.

Temporal-awareness relates time to the usage of tool services. A temporally-aware tool A_i^+ associates (during capture) and enforces (during playback) the notion of relative time to its abstract base machine, i.e., $TS^+(A_i, t) = f(abm(A_i), t)$. A replayable applet A_i* is a temporally-aware tool, applet, or device driver with a well-defined tool service $TS(A_i)$. As temporal-awareness is defined with respect to a tool, replay-awareness is defined with respect to the computer-based workspace spanned by those tools. A workspace that is spanned by replayable applets that comply with our replay-awareness interfaces is said to be replay-aware. These interfaces (see Fig. 2) allow a third-party, our session manager, to orchestrate multiple tools in a workspace as a collective. Replay-awareness allows the capture, representation, playback, and manipulation of multiple tools in the workspace.

Over time, the usage of a tool service is modeled by a temporal media stream, (referred to as *an* $\psi()$*-stream* a), that consists of a time-ordered sequence of abstract base machine operations, i.e., $a = [(a_1 = f_i(), t_1), a_2 = (f_k(), t_2), \cdots, a_n = (f_j(), t_n)]$. Two primitives are used to extend temporal-awareness over an applet stream. The `record()` primitive time-stamps and stores an event. The `replay()` primitive loads and schedules an event. Two additional interfaces are used to fine-tune the performance of the previous primitives: `store()` and `prefetch()`. The `store()` primitive is associated to the `record()`, it provides efficient, media-dependent, storage of stream events. The `prefetch()` primitive is associated to the `replay()`, it provides efficient, media-dependent, prefetching of stream events.

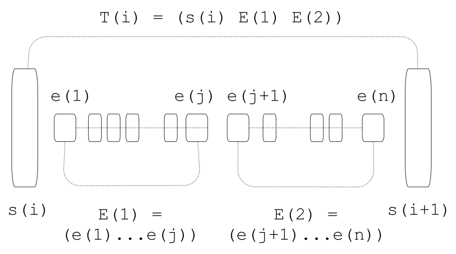

Figure 18. Relationship between events e_i, event sequences E_i synchronization events s_i, and scheduling intervals T_i.

To support interactive, delay-sharing of a session, the **`record()`** and **`replay()`** primitives operate over four different types of events: (1) applet events e_i, (2) state snapshots S_i (3) synchronization events s_i, and (4) branch events b_i. The first two types of events represent intra-applet events, i.e., events that are internally consumed by the applet. The later two are inter-applet events, i.e., events that are consumed by multiple applets. The basic model of a temporal media stream is shown in Fig. 18. The designer of a replayable applet must specify the contents of the intra-applet events (i.e., applet events and state snapshots). Whereas applet events are defined by the selection of an abstract base machine *abm* of an applet; state snapshots are defined by the necessary and sufficient context to support stateful correctness during re-execution on the selected *abm*. The synchronization event $s_i(R_k)$ enforces a **`time equals`** relationship across streams. Whenever an exact temporal ordering across streams in a relationship R_k is needed, the synchronization event $s_i(R_k)$ is used. The frequency and placement of a synchronization request is defined by the application developer. A developer uses a synchronization event s_i to enforce a temporal relationship using relative point synchronization. The placement of synchronization checkpoints must occur at the end or beginning of a well-defined event sequence. A state snapshot S_i contains the full state of an applet and as such state snapshots are defined by the context of an applet. For example, the state of a drawing application is composed of the value for all variables of an applet (items such as pen color, canvas color, canvas size, pen size, etc.). Assume $S_0, ..., S_{i-1}, ...S_n$ is a sequence of state snapshots taken just after synchronization events $s_0, ..., s_{i-1}, ...s_n$. A branch event b_j introduces a fork over the re-execution flow of an $\psi()$-stream (see Fig. 19). Because $\psi()$-streams are stateful, in order to support support editing and annotation of sessions, branch events need to be preceded by a state snapshot event S_k.

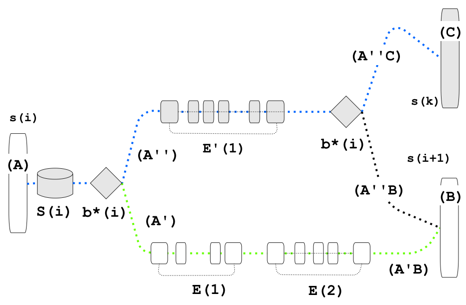

Figure 19. Editing of a re-executable segment.

Fig. 19 further illustrates the supporting model for editing a previously recorded session. Editing of a re-executable segment relies on branching events b_i, state snapshots S_i. The original segment is shown as $A'B$. Interaction reusing the initial state to this segment is shown as $A''B$. Finally, annotation over this segment then coupled with a state jump is shown on by $A''C$. Equivalently,

$$T = \left\{ \begin{array}{l} A'B = (s_A, S_i, b_i(A'), E_1, E_2)s_B \\ A''B = (s_A, S_i, b_i(A''), E'_1, E'_2, b_i(B))s_B \\ A''C = (s_A, S_i, b_i(A''), E'_1, E'_2, b_i(C))s_C \end{array} \right\} \quad (17)$$

$\psi()$-streams have heterogeneous characteristics. Events from different $\psi()$-streams (can) have different characteristics such as:

- are events coarse or very fine grained?
- are events asynchronous? In that case, could they be characterized by an inter-arrival distribution or does a periodical characterization describes them (as for continuous streams).
- are events causally related or are events temporally related?

Such heterogeneity affects the suitability of known policies for synchronization and scheduling of streams.

An event sequence is a series of consecutive stream events. Event sequences must be well-defined. A well-defined event sequence represents an atomic, logical data unit, to the processing (i.e., storage, prefetching, scheduling, synchronization) of an $\psi()$-stream. In a well-defined sequence $E_i = (e_m, ..., e_n)$, intermediary events $(e_{m+1}, ..., e_n)$ lack stateful dependencies to events outside the sequence E_i. Event e_m may be statefully dependent on its immediately preceding state snapshot. Event sequences represent the grain of our synchronization and scheduling mechanisms. This ensures that operations such as prefetching and synchronization are performed at well-defined, stateless, boundaries.

The scheduling interval provides the basic operational unit for our scheduling and synchronization mechanisms. A scheduling interval T_i is composed of one or more event sequences E_i. Scheduling intervals are delimited at each end by synchronization events. Optional state snapshots may precede an event sequence. During playback, a branching events may be inserted immediately after a state snapshot (to support interactivity and editing of a session). Equivalently, the scheduling interval is formally specified by the following regular expression:

$$T_{i+1} = (s_i, \left\{ \begin{array}{l} [S_i], E_i \\ S_i, b_i, \left\{ \begin{array}{l} E_i \cdots \\ E'_i \cdots \end{array} \right\} \end{array} \right\}), s_{i+1} \tag{18}$$

A synchronization relationship $R_k(A_i \leftarrow A_j) : policy$ characterizes the nature of the temporal relationship between A_i and A_j in terms of (1) a master-slave dependency constraint (i.e., the playback of A_j synchronizes to A_i); and (2) a treatment policy for the handling of the dependency constraint (i.e., the schedule of A_j adapts to A_i). For example, the synchronization relationship $R_k(audio \leftarrow gestures) : Policy$, where $Policy$ prescribes "*gestures are to adapt to the unconstrained playback of audio*" represents an end-user characterization of the playback of audio-annotated gesturing. There are two types of synchronization relationships between heterogeneous streams: `causal` and `temporal`. In a causal relationship, our goal is to preserve causality inherent across events of different streams. Causal relationships typically arise between discrete streams (e.g., cut and paste between two $\psi()$-streams) although it can be found too, between a discrete and a continuous stream (e.g., user interactions over a video stream). In a `temporal relationship`, our goal is to preserve relative timing during the streaming across events from different streams. Temporal relationships typically arise between continuous streams (e.g., lip-synchronous video) as well as between a continuous and discrete streams (e.g., audio-annotated gesturing).

A synchronization event $s_i(R_k())$ is used to request point synchronization between scheduling intervals $T_i(A_1), ..., T_m(A_n)$ on $\psi()$-streams involved in a synchronization relationship R_k. A synchronization event $s_i(R_k())$ creates one or more binary dependency constraints $d_i(A_j \leftarrow A_m)$. A dependency constraint $d_i(A_j \leftarrow A_m)$ requires A_j to "*meet*" A_m. A failure of A_j to meet its A_m results in a synchronization treatment *treat* to be applied to A_j. Dependency constraints are asymmetrical (e.g., A_j waits for A_m), which leads to a natural master-slave model for their enforcement. Equivalently, in terms of the global playback times of the synchronization event s_i on each of these $\psi()$-streams A_j and A_m, we get that:

$$d_i(A_j \leftarrow A_m) = \left\{ \begin{array}{l} t_{s_i(A_j)} > t_{s_i(A_m)} : treat(A_j); \\ t_{s_i(A_j)} \approx t_{s_i(A_m)} : nil; \\ t_{s_i(A_j)} < t_{s_i(A_m)} : nil; \end{array} \right\} \tag{19}$$

where *treat* specifies the treatment associated for correcting the playback of A_j for when A_j fails to meet its dependency constraint with A_m.

Different synchronization relationships specify different ways of satisfying the "*meet*" relationship through the *Policy* specification. Let j be the j^{th} synchronization event of an $\psi()$-stream A_a and j^* be its matching (i.e., j^{th}) synchronization event on another $\psi()$-stream A_b. There are three basic meeting policies that specify how j and j^* *ought to meet* during the playback of $\psi()$-streams A_a and A_b. These are:

- `absolute`: $j = j^*$
  that is, the synchronization points j and j^* must be replayed at the same time.
- `lazy`: $j - \epsilon < j^* < j + \epsilon$
  that is, the synchronization points j and j^* can be replayed relatively close to each other.
- `laissez-faire`: $k = j^*$
  that is, the synchronization points j and j^* can be replayed in any sequence.

Not all *meet* policies are the same. Meet policies can be ordered in terms of the strength of their tolerance of asynchrony departure between synchronization points j and j^*. Equivalently,

$$\texttt{absolute} > \texttt{lazy} > \texttt{laissez-faire} \tag{20}$$

Between two $\psi()$-streams j and i, there are three different types of dependency constraints. In a `2-way` relationship, $\psi()$-stream j synchronizes to $\psi()$-stream i and viceversa. That is, whenever an $\psi()$-stream is ahead, it waits for the other. The notation $i : j$ is a short-hand (referred to as a 2-way set) for the representation of a 2-way constraint $[i \leftarrow j, j \leftarrow i]$. In a `1-way` relationship, $\psi()$-stream j synchronizes to $\psi()$-stream i. That is, whenever j is ahead of i, j waits for i. In a `0-way` relationship, $\psi()$-stream j does not need to synchronize to $\psi()$-stream i. That is, $\psi()$-stream j and i are unrelated; so no one waits.

To specify a workspace W, multiple synchronization relationships $R_{k=1\cdots n}$ may be needed. These multiple synchronization relationships need to be reduced into canonical dependency constraints. This problem seems similar to the reduction of functional dependencies during the logical design of a database. The n-ary synchronization handshake is manipulated by the session manager to preserve the relationship $R_k(a \leftarrow b)$. To represent a typical workspace, multiple synchronization relationships are needed. However, not all synchronization relationships have the same synchronization requirements. Similarly, not all $\psi()$-streams have the same playback continuity tolerances. This is particularly true for the playback of heterogeneous streams.

Inside an applet, synchronization relationships are preserved as follows. During design of an applet, an applet's designer needs to specify the synchronization relationships $R_k(a \leftarrow b)$, that the applet holds with respect to other applets in the workspace. To enforce a synchronization relationship $R_k()$, the applet's designer uses a synchronization events as follows. During the record of a session, the applet (a) publishes $s_i(R_k)$ to the session manager; which then introduces the synchronization dependency into the streams in the b set of R_k. The synchronization event $s_i(R_k)$ is recorded as issued by a and accepted by b $\psi()$-streams. During playback, $\psi()$-streams a as well as b publish the synchronization event $s_i(R_k)$ to the session manager. Since multiple relationships can co-exist, it is possible that on a shared synchronization point across relationships, conflicting treatments may arise. To address this, recall that treatments are mechanisms for the implementation of synchronization policies. When conflicting treatments arise, the *treatment* associated with *strongest policy* of $\psi()$-streams in the conflict is selected. The specification of a workspace W produces three tuples: (1) W_A: applets in the workspace, (2) W_R: synchronization relationships between $\psi()$-streams, and (3) W_T: playback tolerances of each $\psi()$-stream.

REFERENCES

1. P.C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *Transactions on Computer Systems*, 13(1):1–31, 1995.
2. K. Birman et al. *The ISIS System Manual, Version 2.0*, April 1990.
3. J. Birnbaum. Pervasive information systems. *Communications of the ACM*, 40(2):40–41, February 1997.
4. M. Cecelia-Buchanan and P.T. Zellweger. Scheduling multimedia documents using temporal constraints. In *Proc. of the 3rd Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 237–249, La Jolla, CA, USA, November 1992.
5. K. Jeffay. On latency management in time-shared operating systems. In *Proc. of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 86–90, Seattle, WA, USA, May 1994.
6. D.R. Jefferson. Synchronization in distributed simulation. *Proc. International Computers in Engineering 4-8 Aug. 1985, publ by ASME, New York, NY, USA*, pages 407–413, 1985.
7. L. Kleinrock. Nomadicity: Anytime, anywhere in a disconnected world. *Mobile Networks and Applications*, 1(4), January 1996.
8. D.A. Kottman, R. Wittman, and M. Posur. Delegating remote operation execution in a mobile computing environment. *Mobile Networks and Applications*, 1(4), January 1996.
9. B. Leiner, V. Cerf, D. Clark, R. Kahn, L. Kleinrock, D. Lynch, J. Postel, L. Roberts, and S. Wolf. The past and future history of the internet. *Communications of the ACM*, 40(2):102–108, February 1997.
10. L. Li, A. Karmouch, and N.D. Georganas. Multimedia teleorchestra with independent sources: Part 1 — temporal modeling of collaborative multimedia scenarios. *Multimedia Systems*, 1(2):143–153, Spring 1994.
11. T. Little and F. Ghafoor. Models for multimedia objects. *IEEE Journal of Selected Areas of Communication*, 8(3), April 1990.
12. N.R. Manohar and A. Prakash. The Session Capture and Replay Paradigm for Asynchronous Collaboration. In *Proc. of European Conference on Computer Supported Cooperative Work (ECSCW)'95*, pages 161–177, Stockholm, Sweden, September 1995.
13. J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, 1986.
14. D.L. Parnas and D.P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, 7 1975.
15. Tristan Richardson, Frazer Bennett, Glenford Mapp, and Andy Hopper. Teleporting in an X window system environment. *IEEE Persnal Communications*, August 1994.
16. H. Schulzrinne. Operating system issues for continuous media. *Multimedia Systems*, 4(5):269–280, October 1996.
17. D.L. Stone and K. Jeffay. An empirical study of delay jitter management policies. *ACM Multimedia Systems*, 2(6):267–279, January 1995.