Flexible Conflict Detection and Management In Collaborative Applications

W. Keith Edwards
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
+1 (415) 812-4405
kedwards@parc.xerox.com

ABSTRACT
This paper presents a comprehensive model for dealing with semantic conflicts in applications, and the implementation of this model in a toolkit for collaborative systems. Conflicts are defined purely through application semantics—the set of behaviors supported by the applications—and yet can be detected and managed by the infrastructure with minimal application code. This work describes a number of novel techniques for managing conflicts, both in the area of resolution policies and user interfaces for presenting standing conflicts in application data.

KEYWORDS: CSCW, collaborative infrastructure, conflict management, Timewarp.

INTRODUCTION
Most collaborative applications involve the shared use of some artifact by a number of users. Infrastructures for supporting such applications must implement mechanisms for dealing with *consistency* in the shared artifact—that is, the degree to which the views and data shared by the participants are the same, and structurally intact.

Some systems support a strict consistency model, in which all participants have exactly the same views and data at all times. Examples of mechanisms for supporting strict consistency include floor control systems and pessimistic locking protocols.

Other systems allow divergence—albeit usually temporary—among replicas. Examples of such mechanisms include optimistic locking [7] and "operational transformation" strategies [6].

In all of these cases, the developers of the infrastructure make an implicit choice about the form of consistency control that will be used by applications built atop the infrastructure. This choice represents some point in the space of consistency versus performance and scalability.

This work presents a new framework for thinking about consistency. In this model, the semantics of the application participate in the definition of consistency and the particulars of how—and whether—consistency will be maintained and managed.

More concretely, this work deals with the issue of *conflicts*—violations of the consistency invariants in a given application. Rather than simply taking the approach that conflicts must be avoided through consistency controls, resolution protocols, and the like, this model treats conflicts as a naturally-arising side effect of the collaborative process. These conflicts may be resolved in any number of ways, or even tolerated and allowed to stand.

Further, since applications have widely-varying notions of what constitutes a conflict, this work accommodates *application-defined* (rather than a priori infrastructure-defined) semantics for describing what constitutes a conflict and how conflicts are managed.

The goal of this work is to address a set of mechanisms that can be used *across applications* to manage "high-level" application-defined conflicts. Put another way, the central question here is how one builds infrastructure, which by its nature must be general, that can detect and manage conflicts which by their nature arise from application semantics.

This work is described in the context of the Timewarp collaborative toolkit [5]. Timewarp is an infrastructure for building applications that permit divergent views of an artifact shared among participants. Users of Timewarp applications have available to them the complete, global history of the artifact, and can move through and edit the multiple, parallel histories of the artifact to manage divergence. Timewarp provides an *explicit* representation of the work of multiple users across versions of an artifact. The history of the artifact itself becomes a shared, globally-available artifact that can be used to mediate collaboration.

While the implementation discussed here, and even the notion of making artifact histories explicit, is particular to Timewarp, the model of divergence that it makes explicit is essentially identical to the *implicit* divergence that exists in many multi-user systems. In such systems, multiple users work with versions of an artifact that may be divergent, and often are later reconciled or merged into a single version. Timewarp presents the problems of versioning and alternate artifact histories in a microcosm—all of these applications deal with the same problems, but reify their models of

operations and conflicts in different ways. The strategies and mechanisms for dealing with conflicts in Timewarp should be "portable" to other collaborative infrastructures.

This paper begins by discussing some goals for this work, based on the perspective of several applications that support rich conflict semantics. From this perspective, we next examine a taxonomy of the types of conflicts that may arise in collaborative applications. This analysis raises several issues that must be addressed by a conflict management system, and motivates our discussion of a particular model for conflict management, and the requirements for a supple infrastructure to support conflict management. We discuss the two major features of our model—conflict detection and management—and describe how these features can be applied to a range of application needs. A number of novel approaches to conflict management are explored. We then examine the particulars of how the Timewarp toolkit implements this model.

PERSPECTIVE, MOTIVATION, AND GOALS

When work started on the Timewarp toolkit, it was clear that the issue of conflict management would be of paramount importance in determining its usability and power. Nevertheless, early versions of the system had little infrastructure support to assist in managing conflicts.

The first application, a structured drawing editor, had a relatively simple conflict model. Still, the code to detect, resolve, and manage conflicts, as well as provide a user interface to these functions, quickly grew to several thousand lines of code that was difficult to manage and more difficult to extend. Worse from the perspective of a toolkit designer, the conflict management code began to "pollute" the relatively clean application programming model.

The situation worsened with the next application, an office furniture layout program. While superficially similar to the drawing tool, this application supported multiple *types* of conflicts. These conflict types, in addition to having very different semantics, have radically different requirements for detection and resolution, as we shall see later in the paper. Complicating the matter further was the fact that some operations in the application could simultaneously cause *multiple* types of conflicts to arise. Dealing with conflicts in complex applications can potentially cause an explosion in code size and complexity, especially when the combinatorial nature of most conflicts is considered.

From experiences with these and other later applications, several goals arose for this work:

- **Simplicity**. The infrastructure should, at all costs, minimize the required work for application writers.
- **Scope**. The infrastructure must support not only conflict detection, but also resolution and user interface issues.
- **Flexibility**. Despite outward similarities, the drawing and layout applications used very different approaches and user interfaces for dealing with conflicts. The infrastructure must support a wide range of application requirements in the arena of conflict management.
- **Modularity**. The mechanics of conflict detection should be separated from conflict handling policy. Thus, application writers should be able to "plug in" different conflict handling policies and have their applications still work. This modularity should encourage experimentation with radically new forms of conflict management.

More specifically, there are a number of concrete issues that must be addressed by an infrastructure to provide flexible management of conflicts. Applications may have widely varying requirements for dealing with conflicts—some applications may be able to tolerate certain types of semantic conflicts; others may not. Some applications in which conflicts may occur may be able to automatically resolve some conflicts without human intervention; others may require manual resolution. Thus our conflict infrastructure must be able to deal with several distinct, specific problems:

- Detecting the presence of runtime inconsistencies within sets of application-supplied invariants.
- Supporting mechanisms for both automatic and manual resolution of conflicts. "Resolution" means doing away with the situation that caused the conflict to arise in the first place.
- Allowing certain types of unresolved conflicts to be tolerated, and others to be disallowed, depending on application semantics and requirements.
- Providing a systematic way of dealing with UI concerns about notification and comprehension of conflicts.

In the next section we examine the types of conflicts that can arise in multi-user applications. Most multi-user applications can support rich models of conflict; understanding how conflicts arise and interact is essential for developing robust conflict management strategies.

A BESTIARY OF CONFLICTS

Syntactic versus Semantic Conflicts

It is useful to make a distinction between two broad classes of conflicts. Dourish [3] classifies conflicts as either *syntactic* or *semantic*. Syntactic conflicts represent inconsistencies that occur below the level of application code; that is, in the toolkit and systems infrastructure itself. Semantic conflicts are inconsistencies that occur above the dividing line between application and infrastructure.

A given application may have completely sound, internally-consistent data structures (that is, have no syntactic conflicts), and yet still expose application-level semantic conflicts to its users. A common example of such a system is a version control tool. Such a tool expects its revision logs to be accurate and internally-consistent, but can expose semantic conflicts when merging disparate file versions to its users.

Conversely, although more rare, applications may be built on an infrastructure that allows—perhaps temporarily—syntactic inconsistencies; these systems may or may not expose application-level semantic conflicts. In fact, in some cases the application itself may not even be "aware" that the infrastructure is managing internal conflicts as it runs. An

example of such an infrastructure is the Bayou system [11], a weakly-consistent distributed data storage system that presents a relational data model to application writers. At any given moment, a Bayou server may contain data that, from the perspective of the application using the data, is not internally consistent. Applications can choose the level of consistency they wish the infrastructure to expose to them.

Although this distinction is a somewhat artificial one—one person's infrastructure is another person's application—it turns out to be useful when considering problems of programmatic interfaces and application support for managing conflicts.

This work addresses facilities for managing *semantic conflicts*. We shall not deal with mechanisms for supporting weak consistency, or epidemic algorithms for obtaining eventual consistency, or locking, or other approaches for addressing syntactic consistency. Further, we shall not address the issue of *how* conflicts arise—whether through weak consistency or network partitioning or merging.

Rather, this work assumes that in most complex collaborative systems, conflicts *will* occur simply because of the semantics of multi-user applications. The specific focus of this work is how to build a systems infrastructure that can support the needs of *applications* to manage the types of conflicts that arise purely because of their own particular semantics. The model here does not necessarily assume that applications require syntactic consistency. Rather it does not address the problems of syntactic consistency at all, and considers them orthogonal to the problems of semantic consistency. Applications using this model may or may not be built on an infrastructure that tolerates syntactic inconsistencies.

Some Classes of Semantic Conflicts

Within the category of semantic conflicts, there are potentially any number of *classes* of conflicts that can arise. Consider a shared drawing application as an example. Suppose that because of the architecture of this application, operations on the shared artifact can become arbitrarily interleaved—whether through weak consistency among replicas, editable timelines *a la* Timewarp, or the merging of two divergent versions of the drawing.

Several types of semantic conflicts can arise in this situation. First, consider the case where an operation draws a new figure on the canvas, and is followed by an operation that moves that same figure. Now, if an operation that deletes the figure is performed between the draw and move operations, a *temporal* conflict will result—the move operation will now refer to a figure which no longer exists in the drawing.

Temporal conflicts arise because of inconsistencies in the up-stream (earlier) and down-stream (later) dependencies in a sequence of ordered operations.

As a second example, imagine an office furniture layout tool that does not allow overlapping figures. In this case, even if the application does not allow a single user to place an object directly on top of another one, operations may still become interleaved in such a way that objects overlap one another, whether through merging or movement toward eventual consistency. This example shows a *spatial* conflict—an application constraint on the placement of objects has been violated.

Both of these classes of conflicts, temporal and spatial, arise purely through the semantics of the particular application. Other applications may not consider such sequences of operations conflicts, or may not even support similar operations at all. Likewise, other applications may have completely new classes of conflicts: *structural* conflicts in a flowchart editor that requires that all nodes be reachable from a root, for example.

These examples by no means represent all types of conflicts that can occur in applications. Rather, the represent several specific types of conflicts that can happen in a given application, and point to the complexities and wide variations that arise in defining application-specific conflicts. Further, they illustrate that any given operation may simultaneously participate in *multiple* classes of conflicts. As described above, the move operation, for example, participates in both temporal and spatial conflicts.

The next section describes the setting in which application-defined conflicts may occur. This setting provides a basis for describing the divergent state of artifacts that are the focus of a collaboration.

ASSUMPTIONS

This work assumes that, for any application, there exists a set of atomic operations that affect the artifacts exposed by the application; these operations are called *actions*. Further, these actions are the *only* way to change the state of the artifact in a way that is significant to other users of the artifact. For example, in a shared drawing tool, the artifact is the drawing itself. The atomic operations on this drawing may include drawing a new figure at specified coordinates, moving an existing figure, or cutting, copying, or pasting figures to and from the clipboard. Other operations that do not *globally* change the state of the artifact—such as setting a zoom factor that only changes the local view—need not be represented in this set of atomic operations.

At runtime, actions are ordered into a directed acyclic graph called a *history*. Each edge represents an action performed by the user, and each node represents the state of the artifact after all upstream actions have been applied. The history represents the complete record of the artifact as it exists across time. For applications that enforce a strict global, serial ordering, the history graph may be simply a straight line of actions. For applications that allow multiple users, perhaps at different sites, to see (or "receive") actions in arbitrary orders, the history graph will be more divergent.

This history graph is conceptually identical to graphs representing message broadcasts among a set of machines, or graphs representing multiple versions of an artifact. The same issues of divergence, conflict, and merging arise in any of these cases, and this model for conflict management is applicable to them as it is to Timewarp.

Any path through the history is called a *timeline* and represents a particular ordering of actions—in essence, a

plausible alternate history of the artifact based on a serial ordering of actions.

For many applications, each user will have a "current node" that represents the "location" of that user in the history. In most applications, the current node of a given user will be at a leaf node in the graph, representing the "cutting edge" of time: all upstream actions will have been seen by that user, and no new, unprocessed actions will have been received. In some applications, however, users may have a current node in the interior of the history. Such applications may allow scanning through alternate plausible timelines in the history.

A *conflict* is defined simply as a special state that exists between any two actions that have been applied, perhaps tentatively, to the artifact. Thus, in our model, when a conflict occurs it is the *operations* in a timeline that are in conflict, not states of the artifact. The model itself assigns no special meaning to the "state of being in conflict" Any semantics of the "conflict state," including how the state arises, is defined purely by the application. Thus there are no "intrinsic" conflicts in this model; *all* conflicts are defined as violations of some application-supplied semantics.

The model here is motivated by other work in the field, including Prospero [3], GINA [2], and WeMet [10].

In the next section we address mechanisms for detecting conflicts that occur in this setting of ordered, atomic operations.

DETECTION OF CONFLICTS

As stated earlier, this work deals with types of conflicts that arise purely from application semantics. An important issue, then, is how we can build an infrastructure that can detect these conflicts, when only application code "understands" that the conflicts exist? How can application semantics inform the infrastructure to detect types of conflicts such as those seen earlier (spatial, temporal, and so on)?

Just as the set of actions provided by an application define its semantics of behavior, these same actions also define the semantics of determining when a conflict exists. Since the notion of whether a conflict exists arises solely through the semantics of what particular actions do, the actions set the conflict *detection* semantics for the applications, since only they can "know" how their behavior can create the presence of conflicts.

This section describes our detection model, based on using application-provided actions to define conflict relationships. After the model is discussed in the abstract, we detail a particular implementation of it in the Timewarp toolkit.

Conflict Sets

The combination of some types of actions in the history graph may potentially cause conflicts to arise; other types of actions may be "immune" to conflicts—no matter how they are added, or in what order, they will never cause a conflict to arise. Types of actions that may potentially cause conflicts among other actions in a limited group are said to define a *conflict set*. A conflict set is simply a group of *types* or *classes* of actions that have the potential to generate conflicts with each other. The presence of an action class in a conflict set is statically-defined, in the sense that it will not change as long as application semantics do not change.

These sets correspond to the various types of conflicts that may exist in an application. For example, in a drawing application, a *temporal* conflict set may include Cut actions, as well as any actions that refer to or modify the state of objects in the drawing, including Move actions. These action classes are grouped together because the addition of any one can interact with other actions in the set to *potentially* cause a conflict to arise, based on the order of the actions. So for example, if a Cut action is placed in a timeline before some other action Moves the object which is cut, a temporal conflict will exist. Likewise, if a Move is added after a Cut is in place, a temporal conflict will exist.

A given action class may simultaneously exist in multiple, overlapping conflict sets. In a flowchart editor, a Move operation may also have the potential to violate structural consistency—perhaps by dislodging an object's connections to its neighbors. In this situation, the Move action would exist in both the temporal and the structural conflict sets.

In essence, a conflict set is simply a means of grouping together types of atomic operations that can cause conflicts with each other based on the semantics supplied by the application. Action classes can exist in multiple conflict sets simultaneously.

Since, by definition, actions can only cause conflicts with other actions in their same sets, sets allow us to partition the space we must consider when searching for conflicts.

Conflict Roles

Within a conflict set the comprised action classes are grouped into *conflict roles*, indicating the part the actions play in the set. So, for example, in the temporal conflict set in the drawing application, all actions that remove drawn objects (such as Cut) might be grouped together in a role; similarly, all actions that modify existing objects (such as Move) might play a similar role in the set and would thus be grouped into a single conflict role. Actions in the same role behave similarly with respect to the set they are in.

Roles partition the functions of actions within a set. For purposes of detection, we need only consider the roles an action plays in its sets, not the semantics of the action itself.

An Example Conflict Hierarchy from a Layout Application

Figure 1 shows the conflict sets and roles for the layout application mentioned earlier. These are simply the conflict relationships for a particular application; other applications may have different types of conflicts, and hence very different relationships expressed through their sets and roles.

Here, the sets and roles are presented as a hierarchy, although conceptually each could exist independently. (The description here matches the implementation used by Timewarp; see the Implementation section for more details). If an action is not a member of any of these sets then it is "immune" from causing any conflicts, no matter how or in what order it is inserted into the history graph.

There are two first-level sets: `Temporal` and `Momentary`. The `Temporal` set represents all types of conflicts that cause

inconsistencies *across* a timeline; `Momentary` conflicts, which include both `Spatial` and `Structural` conflicts, are only based on the instantaneous state of the artifact and do not affect all of a timeline (this distinction, and the implications of it, will be explained more thoroughly in the next section).

Within the `Temporal` set there are three roles that actions may play. The `dependsOn` role indicates that the action depends on some other action being earlier in the timeline. The `dependable` role indicates that the action may be used as a target of a `dependsOn` relation. The `seversDependsOn` role indicates that the action may break, or sever, any downstream `dependsOn` references to an upstream `dependable` action.

Within the `Spatial` subset of `Momentary`, the `existsAt` role indicates that the action it is associated with produces or modifies some figure in the drawing so that it "exists at" a certain spatial location. The `seversExistsAt` role indicates that the action associated with it removes an object created by another action.

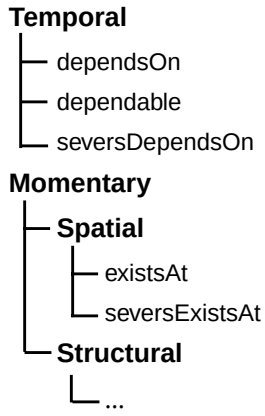

FIGURE 1: Hierarchy of Conflicts for a Layout Application

Table 1 shows the actions that constitute the layout application. Note that most actions participate in several different conflict sets. The `CreateObject` action participates in both the `Temporal` and `Spatial` sets. It is `dependable` because it creates an object that may later be referred to by another action; it also `existsAt` since it produces a figure in the layout which has a physical position and size. The `MoveObject` action `dependsOn` a previous object that it will move. It undoes that object's old position (via `seversExistsAt`) and relocates it (via `existsAt`). `CutObject` both `dependsOn` the existence of an upstream object that it will cut, and, via `seversDependsOn`, indicates that any downstream objects that depends on the deleted object are now in conflict with this `CutObject` operation.

When an application writer creates a new application, he or she defines the set of actions that represent the allowable operations in the application. At the same time, the application writer defines the conflict relationships between these applications by designing a group of conflict sets and assigning the actions into it. These relationships are used by the detection machinery to identify conflicts automatically when they occur.

Action	Set:Role
CreateObject	Temporal:dependable Spatial:existsAt
MoveObject	Temporal:dependsOn Spatial:existsAt Spatial:seversExistsAt
CutObject	Temporal:dependsOn Temporal:seversDependency Spatial:seversExistsAt
PasteObject	Temporal:dependsOn Temporal:dependable Spatial:existsAt

TABLE 1: Conflict Roles of Actions in the Layout Application

Gathering Conflict Candidates

At an abstract level, the detection process uses a source action and compares it repeatedly against other target actions to see if they are in conflict. The first phase of the detection process is to gather all of the potential conflict *candidates*. These are the other actions in the history graph that can, in any way, factor into the decision of whether conflicts exist.

For a given action, its candidates are the other actions from some sub-region of the history graph that are members of the same sets as itself. The *particular* sub-region that is considered depends on the sets in which the source action is a member. Consider, for example, temporal conflicts. When a new Cut operation is inserted into the history graph at a given point, the conflict detection machinery must consider all paths that pass through the new action, looking for violated downstream dependencies, to determine if a conflict exists. This is because temporal conflicts, by their very nature, are inconsistencies affecting downstream actions.

See Figure 2 for an example of such a conflict. It is possible that the insertion of the action will cause a conflict in one path, but not another. For this reason, all actions in all timelines passing through the insertion point are considered candidates for temporal conflicts. Because such conflicts are *situated* in the timeline itself, they corrupt the entire timeline they exist in, from the point of the earliest conflicting action. In the figure, the entire upper path from the point of insertion of the Cut operation is in an inconsistent state.

Spatial conflicts do not have this unbounded property, however. The determination of whether a spatial conflict exists depends only on the *current* state of the artifact, which may be represented as the sequence of actions leading up to the insertion point. Downstream actions do not play into the decision of whether a conflict exists. So for spatial conflicts, all actions up to the insertion point are considered to be candidates for conflict.

Figure 3 shows an example of such a conflict. Action 1 causes a figure to be drawn at coordinates *X,Y*. Action 2 moves a different figure to the same coordinates. This action

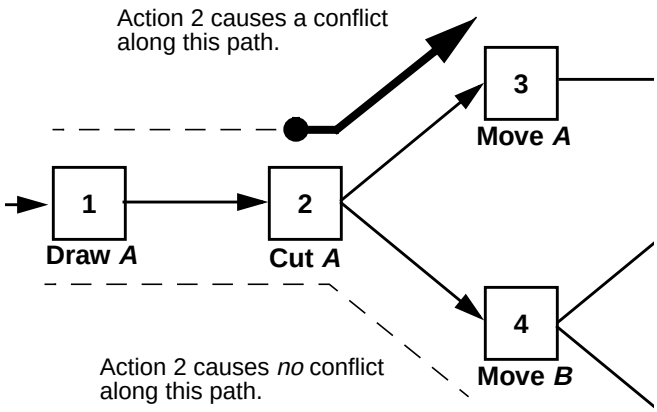

FIGURE 2: Temporal Conflicts Invalidate Some Downstream Timelines, But Not Others

causes a conflict to occur, based only on the state of the artifact up to the point of the insertion—by examining only upstream actions, the system can determine that two objects are in the same position. Further, note that such *momentary* conflict exist only for a bounded span of a timeline. Action 3 moves the second figure to a new, non-conflicting location. Thus, unlike temporal conflicts in which an entire timeline is corrupted, momentary conflicts are confined to a limited area of a timeline.

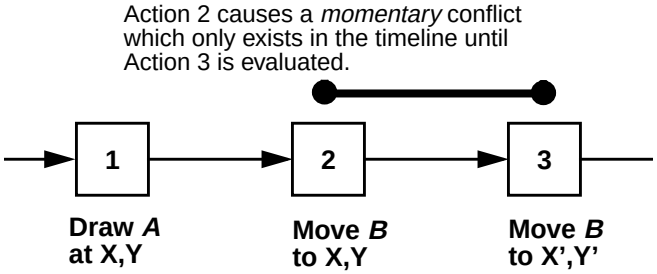

FIGURE 3: Spatial Conflicts Span a Region of a Timeline

Since temporal and spatial conflicts involve different sets of actions, we must use different gathering strategies for each (and potentially different strategies for all the types of conflicts in an application). Since an action may be a member of multiple conflict sets, gathering is performed separately for each set and candidates are then passed to the identification phase of detection.

Identification. After the set of conflict candidates has been gathered, actual identification of conflicts is performed. The identification process for a given set is only "handed" the gathered candidates for that set, even if the action in question is a member of multiple sets. Identification of conflicts is separated by set to make the task of identification of conflicts for a particular set independent of all other sets.

The process of identification simply iterates through the gathered candidates. For each candidate, a set-specific predicate function is evaluated to determine if a conflict exists between the source and candidate. If a conflict does exist, implementations may cache the pair of conflicting actions, along with information about the conflict set.

This phase only accumulates *actual* conflicts from the set of *potential* conflicts gathered earlier. No action is taken to resolve or disallow any conflicting actions at this point.

Requirements for the Application Writer

Despite the complexity of the detection process, adapting an application to work in this model requires fairly little effort. At a minimum, the application writer must define the types of conflicts that can exist in the application, and assign the various operations supported by the application to conflict sets and roles. The infrastructure needs to be able to query actions for a list of the sets they are members of, and their roles in those sets. The particulars of how set and role information is associated with actions can vary from implementation to implementation. Timewarp implements the model by requiring actions to implement Java interfaces that represent their conflict roles (see the section on Implementation). Other implementations may use multiple inheritance, or simply a set of type flags. This definition phase is a static (compile-time) process that will not change as long as the semantics of the application do not change.

Next, the application writer must define per-set gathering and identification functions for each conflict set in the application. In most cases, one of the two types of gathering described earlier—whole-path gathering or current-state gathering—will be applicable, and thus gathering functions can be reused. The identification function is simply a predicate that is passed two actions in the same set and returns either true or false indicating whether they are in conflict. Typically these functions are on the order of a few dozen lines of code total.

The model partitions the detection process both procedurally—into gathering and identification phases—and structurally—by grouping actions into sets and roles. This partitioning greatly reduces the "cross-talk" among types of conflicts, and minimizes and simplifies the amount of application code required.

Once this process of defining the conflict relationships in an application has been completed, an infrastructure using this model can automatically detect conflicts that arise among the actions in an application. Further, as we shall see, our infrastructure will also be able to automatically manage any conflicts that are detected.

MANAGEMENT OF CONFLICTS

Handling conflicts in an application has two aspects. First, the description of what constitutes a conflict is statically-defined and rooted in the semantics of a particular application. These semantics are made manifest as actions, and hence actions are statically grouped into conflict sets and roles to denote their relationships to one another. The previous section on detection of conflicts covered this process.

The second aspect of conflict handling, however, is independent of the particular semantics of the set of actions used by a given application. Unlike detection, this aspect is

dynamic—it indicates what to *do* with conflicts once they have been detected. This process is run-time behavior—unlike the compile-time definition of conflict relations—and can be made largely independent of the semantics-based detection mechanisms.

Our model codifies this distinction by separating conflict detection from the behavioral aspects of dealing with a set of detected conflicts. The behavioral aspects are dictated by a *conflict policy* that embodies the semantics of conflict management at run-time, separate from detection.

One benefit of making the distinction between action-based detection and policy-based management is that the conflict policies can be made independent of the semantics of the conflicts they are asked to manage. So, for example, an application may come with a set of "pluggable" conflict policies that can be swapped in and out. Any of these policies would be capable of managing *any* set of conflicts detected as described above. If they so desire, however, application writers can create policies that have intimate knowledge of the semantics of particular classes of conflicts.

Conflict management is performed in response to detected conflicts. When a new action is created but before it is installed in the history, detection is done to determine whether the tentatively installed action would cause any conflicts to arise. If conflicts would be caused by this new action, then the conflict management system is invoked to deal with the detected conflicts.

Conflict policies are associated with particular conflict sets. Typically an application will associate a policy with each conflict set defined by the application.

In our model there are several different aspects of conflict management that policies provide:

- **Resolution**. If a tentative action causes an inconsistency, the infrastructure can provide support for either automatic or manual resolution of the inconsistency.
- **Tolerance**. After any optional resolution is performed, the policy can decide whether the tentative action should indeed be inserted into the graph. This model allows conflicts that have not been resolved to still be tolerated in the graph, at the discretion of the application.
- **Interface**. Once inserted into the graph, actions that cause unresolved-but-tolerated conflicts may influence the user interface of the application. The infrastructure provides "hooks" for allowing applications to trigger user interface changes when conflicting data is presented.

The conflict management policies can manage *any* detected conflict, regardless of the conflict set it originated from. Policies define how—and whether—resolution will be performed, whether standing conflicts will be tolerated, and interface changes that will be associated with conflicting on-screen objects. Conflict policies have complete access to the history graph and can make arbitrary changes to it (perhaps recursively causing new conflicts). We will now examine these aspects of conflict management in detail.

Conflict Resolution

When an action is tentatively installed and conflicts are detected, the system may attempt to resolve them, thereby removing any inconsistencies that would be caused. Resolution is accomplished by handing the identified conflicts to the policies associated with any sets that identified conflicts. Of course, since actions may be in multiple sets, if these sets are mapped to different policies each policy may participate in the resolution process.

There are a number of common resolution strategies that can be implemented by conflict policies. The simplest strategy is to simply provide the user with the option of not performing the new, conflicting operation. If the operation is undone before being actually inserted into the history, no conflicts will be caused.

Policies may, however, implement other, more complex strategies for resolving conflicts. Some policies may implement these strategies as automatic resolution procedures, which run without user intervention. Others may interact with the user to "tune" the resolution process.

The Explosion Strategy. This strategy lets computation proceed at the cost of history complexity. At any point where a conflicting action exists in the history, the history is forked into multiple downstream paths. Each of these paths represents an alternate in which conflicts are avoided by selectively removing actions involved in a conflict.

Figure 4 shows such an example, where an inserted Cut operation would conflict with a downstream Move. The history is forked into two branches. The first branch "favors" the Cut by letting it stand and removing its conflicting partner, the Move. The second branch does the opposite: the Move is favored and the Cut is deleted.

The explosion strategy resolves conflicts by providing *both* (or multiple) valid interpretations of the state of the artifact when an inconsistency is encountered.

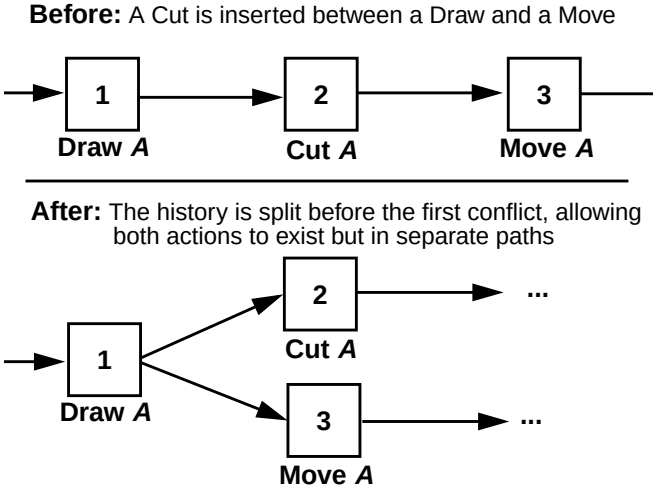

FIGURE 4: The Explosion Strategy

The Promotion Strategy. This strategy "promotes" actions that depend on upstream results into actions that can exist without reference to any upstream information.

Figure 5 shows an example of promotion. Here, a figure has been copied to the clipboard, and then pasted. If the figure is removed before being copied, an inconsistency exists because there is no object to copy to the clipboard, and hence the paste operation cannot proceed.

In the figure, promotion is used to replace the Paste operation, which depends on the existence of upstream actions, into a "stand-alone" Draw operation that has no such dependencies. In effect, the dependency is severed and the conflict is undone.

Such a strategy may not be effective for all situations. But in cases where the Paste was performed by the user essentially as a "short-cut" for a Draw, promoting conflicting Paste operations to Draw operations preserves the intended meaning of the user.

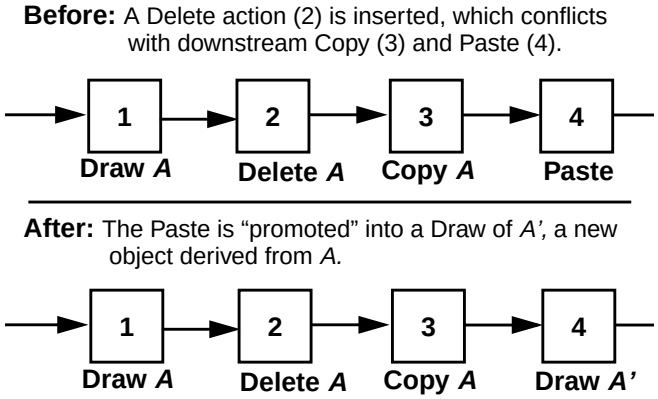

FIGURE 5: The Promotion Strategy

The Recursive Acceptance Strategy. One obvious option for resolving conflicts is, rather than disallowing the insertion of the new action, remove any downstream actions that conflict with it. Of course, the removal of these actions may itself cause new conflicts to be created, which may be resolved.

The recursive acceptance strategy uses this approach. When a conflict is detected, the user has the option of removing either the source action, or any conflicting targets. When a target is "removed," the system actually creates a new tentative timeline and recursively executes the conflict detection code to determine what new conflict may have been created. At this point, the resolution policies for these new conflicts is executed. The process continues until the user has either reconciled the timeline or allowed the conflicts to stand. At any point the user can "back out" to undo an earlier conflict decision.

The Quantum Uncertainty Strategy. Quantum uncertainty is not so much a pure resolution strategy as a combination of tolerance and user interface ideas. In quantum uncertainty, inconsistent operations are allowed to exist. All such operations are said to be in an "uncertain" state, and have a numeric "uncertainty" value associated with them. This value is recalculated whenever a change is made to the history graph, and represents an index of the validity that the action seems to have, based on the other actions around it.

In quantum uncertainty, the accretion of actions in a history is used to posit a likely interpretation of inconsistent data. Thus, a set of actions that reinforce one interpretation of a conflict lessens the uncertainty of one of the actions participating in the conflict, while increasing the uncertainty of the other.

Figure 6 shows an example of quantum uncertainty. Here, we see our by-now classic Draw-Cut-Move inconsistency. After the insertion of the Cut, both the Cut and Move operations have uncertainty values of 0.5 indicating that they are equally uncertain. In the second graph shown, a sequence of Moves and Copies of the supposedly-cut object are appended to the timeline. These actions all "ignore" the interpretation of the conflict that favors the Cut operation, so they lessen the uncertainty of the Move, and all other actions that conflict with the Cut, at the expense of the Cut. In effect, these later operations reinforce the interpretation that the Cut should be ignored. Other divergent paths may favor the Cut, in which case the situation would be reversed in those paths.

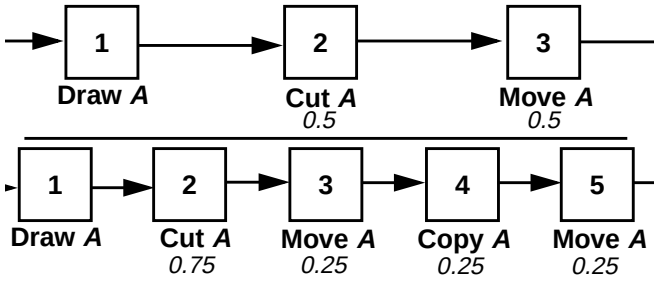

FIGURE 6: The Quantum Uncertainty Strategy

Resolution Summary. All of these strategies show the degree of control the conflict management system has over the history of the artifact. New paths can be created in the graph, and actions can be removed and rearranged. Because of the ability to tentatively insert actions and detect resulting conflicts, the conflict manager allows "what if" scenarios (such as shown by the recursive acceptance strategy) in which the user can iteratively resolve cascading conflicts, and yet still back out without damaging the artifact.

Conflict Tolerance

An important trait of the Timewarp conflict model is that it allows applications to manage *tolerated* conflicts. That is, some applications may decide that certain types of conflicts need not be resolved, but can be allowed to exist in a timeline. A common reason for tolerating conflicts is to ease users' burden—if the users of a tool understand the intended state of the shared artifact and are satisfied with it, there may be no reason to force them to go through a series of steps to manually resolve any existing inconsistencies. Work can proceed at its own pace, and users can elect to resolve conflicts later, if at all.

For example, in the layout application, conflicts often arise because two users working independently merge disparate action paths into one shared state. The merge may cause certain objects to overlap, violating spatial constraints in the application. Rather than forcing the users to readjust the

entire layout to do away with the inconsistencies, the system allows the inconsistencies to remain, but flags them in the interface (more on this later).

Whether or not a class of conflicts will be tolerated depends solely on the conflict policy in effect for that class of conflicts. In the Timewarp implementation, determination of tolerance occurs after the optional resolution step; the policy returns a boolean value indicating whether the tentatively inserted action should be installed in the history graph, or should be discarded. Tolerated conflicts are cached so that they can be used to effect interface changes if desired.

Conflict Interfaces

Allowing standing conflicts to exist in an application raises questions about how they should be presented to users. Our infrastructure must be able to allow application code—especially the application's user interface code—to collude with the conflict management system to systematically modify the application interface to reflect conflicts.

Interface Examples. One of the goals of this work is to explore rich interfaces for presenting conflicts. Several interfaces for interacting with standing conflicts have been developed. In the layout application, spatial conflicts are handled by a policy that allows the conflicting, overlapping objects to co-exist, but renders them transparently and slightly blurred via a Gaussian blur ([5] contains an image of this effect). This technique provides an immediately obvious cue that the state of these objects is somehow "different" than their neighbors. The fact that the objects are physically on top of one another and yet can both be seen strengthens the perception of the cue as indicative of spatial conflict.

In the drawing application, temporal conflicts are handled by quantum uncertainty. Drawing of conflicting objects is modified so that the transparency of an object represents its degree of uncertainty. Fully certain, unambiguous objects are rendered fully opaque. Other objects, as their uncertainty increases, gradually fade out over time.

Other policies may annotate drawn objects that are involved in conflicts with controls that allow the user to pop up a panel of information about the conflicts they are involved in.

Interface Implementation. The interface of any application is, by necessity, dictated by the user interface toolkit being used as well as the semantics of the particular application. This requires that particular *implementations* of our conflict model know about and support a particular UI toolkit. Still, the conflict management infrastructure can provide generic mechanisms for supporting a range of application needs given a particular UI toolkit.

Our model distinguishes between basic conflict policies and *drawable conflict policies.* Drawable conflict policies are policies that know how to interact with the rendering system of a particular UI toolkit. Drawable conflict policies add a number of new behaviors over simple conflict policies; these behaviors are used by application code to allow some of the application's drawing behavior to be delegated to the policy.

Whenever conflicts are detected along a given path, they are stored and indexed by the particular sets they represent. When application user interface code is about to draw an on-screen object that is a part of the shared artifact—and, hence, may potentially be the product of actions that are involved in standing conflicts—the application may delegate drawing to the drawable conflict policies that affect the object.

Again, the particulars of how this delegation is done depends on the particular toolkit in use. The current implementation of Timewarp provides a drawable conflict policy infrastructure for interacting with the SubArctic GUI toolkit [8], but the general concepts are portable to other UI toolkits, albeit perhaps with more work for implementors.

The infrastructure provides a mapping from on-screen objects to the actions which have affected them. Once this mapping has been created, applications can inquire to the conflict management system about whether a particular on-screen object should have its drawing delegated to one or more drawable conflict policies. If the conflict management system indicates that drawing should be delegated, the application passes the on-screen object into the conflict management system for drawing. The conflict management system then identifies all of the drawable conflict policies that are involved in conflicts for the object. These policies are applied—pipeline fashion—to the drawing process for the object. The final result is that each policy has a chance to impose its own policy-specific drawing modifications to objects involved in conflicts.

This mechanism addresses the integration of conflict management with a UI toolkit, and is extensible to new policies, new applications, and new UI toolkits—it requires only the cooperation of the set of actions used in the application, and the particular conflict policy used to manage the interfaces of conflicts among these actions. No changes are required to the basic conflict management system, and the conflict management code does not have to understand the particulars of a given UI toolkit; only the particular drawable conflict policies in use are required to be able to "speak" to a particular UI toolkit.

The next section discusses the particulars of the conflict interface implementation provided by Timewarp, along with other implementation details.

CONFLICT IMPLEMENTATION IN TIMEWARP

In Timewarp, the ordered atomic operations model is realized as a set of concrete classes, in the object-oriented sense. Actions are classes that represent the atoms of behavior in a given application; particular action classes are provided by application writers when they build their applications. When a new operation is performed, a new action instance is created to represent it, and is installed in a history graph that provides an explicit representation of the parallel timelines of the shared artifact.

Timewarp uses an open implementation [9] approach: the infrastructure "speaks" in terms of actions, generically. Applications are built by extending these actions through subclassing, and then "pushing" them back into the infrastructure, thereby inserting their own semantics into the toolkit framework.

Conflict sets and roles are represented as a hierarchy of interfaces that can be multiply inherited by the actions. So application writers, when creating the actions that constitute their application, declaratively assign these actions to a number of interfaces representing the roles of a particular set. The use of interfaces here provides a declarative notational convenience and strong compile-time type safety.

All of the infrastructure (non-application) code for detecting and managing conflicts is localized in the `ConflictManager` class. This class uses information provided in the types of the actions defined by the application writer (accessed via reflection), as well as explicitly-provided application code such as detection predicates, to implement the conflict model described here.

The current implementation provides a number of drawable conflict policies that interact with the SubArctic GUI toolkit. These policies leverage the "drawing isolation" features of that toolkit [4] to provide convenient pipelines of drawing operations that represent the effects of multiple drawable conflict policies. UI-specific code is not present in the infrastructure, with the exception of the particular drawable conflict policies where it is isolated.

The Timewarp system is implemented entirely in the Java programming language [1]. Currently, the framework itself is approximately 15,000 lines of code. The entire conflict subsystem, along with the set of "pluggable" policies described above, represents 2,000 lines of the total. A number of applications have been created using this system. The total number of lines devoted to conflict management in each of these applications averages only 60 lines of code, almost exclusively in the detection protocols.

SUMMARY AND FUTURE DIRECTIONS

This paper has described a comprehensive model for thinking about the detection and management of semantic conflicts in applications. These conflicts can arise in any number of application genres, from distributed systems to version control to collaboration. The model here supports rich definition of conflicts, including the ability to define multiple simultaneous conflict categories. Further, the model provides a framework for the resolution and tolerance of detected conflicts, and can interact with application code to produce novel interface effects based on standing conflicts.

This model has been implemented as a part of the Timewarp collaborative infrastructure. While this toolkit provides significant capabilities to application writers for defining and managing conflicts, the amount of work required by application writers to participate in conflict management is minimal. Typically only a small amount of code must be written to interact with the conflict subsystem.

There are a number of future directions for this work. An obvious area is the investigation of more forms of conflict resolution. The forms described here, while novel, do not capture the entire space of conflict resolution. A second area of focus is on conflict interfaces. Our current model only supports *output* changes in response to standing conflicts. We would like to also be able to modify the *input* to on-screen objects based on the conflict policies in effect. A final area for future work is on declarative specification of detection protocols. In the current system, detection is implemented procedurally by the application writer in the form of a detection predicate. We are investigating a declarative implementation based on constraints, which may provide a more natural way to define conflict relationships.

ACKNOWLEDGEMENTS

Thanks to Paul Dourish, Beth Mynatt, and Ian Smith for their contributions in reviewing and editing this paper.

REFERENCES

[1] Arnold, K., and Gosling, J. *The Java Programming Language.* Addison-Wesley Co., Reading, MA, 1996.

[2] Berlage, T., and Genau, A., "A Framework for Shared Applications with a Replicated Architecture," *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, Atlanta, GA, November, 1993.

[3] Dourish, P. *Open Implementation and Flexibility in Collaboration Toolkits.* Ph.D. dissertation, Computer Science Department, University College, London, June, 1996.

[4] Edwards, W.K., Hudson, S.E., Marianucci, J., Rodenstein, R., Rodriguez, T., Smith, I. "Systematic Output Modification in a 2D User Interface Toolkit," *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, Banff, Alberta, Oct. 1997.

[5] Edwards, W.K., and Mynatt, E.D., "Timewarp: Techniques for Autonomous Collaboration," in *Proceedings of the ACM Conference on Computer-Human Interaction (CHI).* Atlanta, GA, March, 1997.

[6] Ellis, C.A., and Gibbs, S.J, "Concurrency Control in Groupware Systems," *Proc. ACM SIGMOD Conference on Management of Data,* June 1989.

[7] Greenberg, S., and Marwood, D. "Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface." *Proc. ACM Conference on Computer-Supported Cooperative Work (CSCW).* Chapel Hill, NC, October, 1994.

[8] Hudson, S., and Smith, I., "Ultra-Lightweight Constraints," *Proc. ACM Symposium on User Interface Software and Technology (UIST).* Seattle, WA, November 1996.

[9] Kiczales, G., "Beyond the Black Box: Open Implementation," IEEE Software, January, 1996.

[10] Rhyne, J.R., and Wolf, C.G., "Tools for Supporting the Collaborative Process." In *Proc. ACM Symposium on User Interface Software and Technology* (UIST). Atlanta, GA, November, 1992.

[11] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.