Resolving Concurrent Interactions

Anand Natrajan, Paul F. Reynolds, Jr.
Department of Computer Science, University of Virginia
{anand, reynolds}@virginia.edu

Abstract

Serialization, the traditional method of resolving concurrent interactions, is often inappropriate; when interactions are dependent on each other, other policies for resolving them may be more suitable. We use semantic information to help categorize common interactions encountered in the modeling and simulation domain. This categorization enables us to suggest reasonable policies for resolving the effects of concurrent interactions.

1. Introduction

One of the most significant challenges facing the simulation community is Multi-Representation Modeling (MRM) — the joint execution of multiple models of the same object or process [8]. The crux of the challenge is resolving concurrent interactions on the representations in the different models [17]. Many systems either serialize concurrent interactions or avoid them by restricting the interactions that can co-occur. However, serialization and avoidance are insufficient for resolving the effects of concurrent interactions in the general case. Other solutions, such as accumulating, delaying or ignoring some or all interactions may be more suitable. We describe a new approach that categorizes interactions by augmenting them with a small amount of semantic information in order to resolve them more appropriately.

For effective MRM, the effects of dependent concurrent interactions must be resolved meaningfully. Often, concurrent interactions may have dependent effects, for example, precluding or enhancing the effects of one another. Traditionally, the effects of concurrent interactions have been resolved by serialization, in which the interactions are ordered arbitrarily. However, serialization is often inappropriate because it isolates interactions whose effects must be applied concurrently. Other policies, such as combining or ignoring some or all interactions, do not isolate concurrent interactions and may be more suitable for resolving any dependent effects.

We present a taxonomy of interactions and show how to classify interactions. We assume that MRM designers can understand the semantics of interactions in their application well enough to classify them and formulate policies for resolving them. We present example policies for resolving concurrent interactions. Our taxonomy enables a designer to choose appropriate policies for resolving concurrent interactions.

2. Interactions

An *interaction* between entities is a communication that causes a change in their behavior. Entities in a model communicate with one another or influence one another by interacting. An entity changes the behavior of another entity by means of an interaction. Interactions are fundamental to a useful model because they connect it to its environment. We regard a communication between any two entities as well as changes an entity makes to its own state in our definition of interactions.

When the changes caused by an interaction are applied to attributes in entity representations, the interaction takes effect. A *sender* is an entity that initiates an interaction while a *receiver* is an entity to which an interaction is directed. The *effects* of an interaction are the changes caused by the interaction to the sender and receiver.

Interactions may be *concurrent*, i.e., they may occur during the same time-step. Simultaneous interactions, i.e., interactions occurring at the same time, are concurrent interactions, although the converse is not necessarily true. In a modeling context, we cannot distinguish simultaneous interactions from merely concurrent interactions.

Concurrent interactions may be dependent. A *dependent interaction* is one whose effects are predicated on the occurrence of another interaction. An *independent interaction* is not dependent on any other interaction. For example, two interactions may be related by cause and effect, i.e., one interaction causes the other. The former interaction is independent of the latter, but the latter is dependent on the former. Concurrent interactions may be dependent solely on account of their concurrence, i.e., if the interactions were not concurrent, they would be independent.

A system that permits concurrent interactions requires a policy to resolve any dependencies among interactions and a mechanism to implement the policy. The traditional policy for resolving the effects of concurrent interactions is serialization.

3. Serialization

Serialization, the traditional policy for resolving concurrent interactions, involves applying their effects in sequential order, i.e., one after another. In serialization, concurrence is resolved by ordering or interleaving concurrent transactions appropriately. Serialization preserves *isolation*, which is one of the ACID properties for database transactions [10].

Serialization has been chosen as a policy for resolving interactions in database systems because it satisfies clients' expectations of isolation yet permits concurrent transactions [16] [5]. Isolation assumes that client interactions are not predicated on one another, i.e., they are independent of one another. Serialization isolates client interactions.

Some researchers have proposed policies that relax or extend serialization yet maintain isolation [6] [3]. Some of these policies require semantic analysis in order to increase concurrence [9] [20] [2]. In general, serialization is considered correct but too strict, and alternative criteria relax or extend serialization in order to permit increased concurrence [4] [13] [14] [12] [19][*]. Moreover, isolation of transactions is considered a desirable property of database systems. Next, we discuss situations where isolation may be undesirable.

4. Abandoning Isolation

For some applications, the system must not isolate concurrent interactions since they may be dependent on one another. Serialization and alternative policies that relax or extend serialization isolate interactions. Therefore, they cannot be correct policies for resolving the effects of dependent concurrent interactions. Correct policies for these interactions must not isolate the interactions.

In the following examples, *not* isolating concurrent interactions, i.e., abandoning isolation, enables resolving their dependent effects correctly. Consider entities `A` and `B` that change an attribute `v`. Consider two concurrent interactions: `A.write(v, …)` and `B.write(v, …)`. A sequential order for these interactions could be `A.write(v, …)` followed by `B.write(v, …)` or `B.write(v, …)` followed by `A.write(v, …)`.

In a model of a billiards table, `A` and `B` could be ball entities and `v` could be the velocity of a ball. The two interactions could be `A.write(v, δv`$_A$`)` and `B.write(v, δv`$_B$`)`, where δv_A is a change in `v` caused by `A` and δv_B is a change in `v` caused by `B`. The correct policy to resolve these two interactions is to change `v` by the vector addition of δv_A and δv_B. Serializing these interactions may be incorrect for a number of reasons as discussed below. Let $\oplus$ denote vector addition. Let v_1, v_2 and v_3 be three possible outcomes of adding δv_A and δv_B to the original value v_0 of the velocity `v`.

$$v_1 = (v_0 \oplus \delta v_A) \oplus \delta v_B$$
$$v_2 = (v_0 \oplus \delta v_B) \oplus \delta v_A$$
$$v_3 = v_0 \oplus (\delta v_A \oplus \delta v_B)$$

The parentheses show the order in which the interactions take effect. v_1 and v_2 are computed by serializing the two interactions, whereas v_3 is computed by combining the two interactions before applying them to `v`. Mathematically, $v_1 = v_2 = v_3$. However, when executing a model, the results of these orderings can differ. For example, δv_A and δv_B may be so small that adding them to v_0 individually does not change `v`. However, δv_A and δv_B combined may be sufficient to change `v`. In such a case, $v_1 = v_2 \neq v_3$. As another example, δv_A and δv_B may overcome the inertia of the entity with velocity `v` when combined, but not individually. Finally, suppose a display process P continuously plots the trajectory of the ball with velocity `v`. If `v` changes to v_1 or v_2, P will plot two changes, whereas if `v` changes to v_3, P will plot only one change. The former change causes P and `v` to be temporally inconsistent. v_1 and v_2 are computed by serialization, whereas v_3 is computed by combination. Here, combination is a more meaningful policy than serialization.

In a model of an autonomous agent, `A` could be a planner that pre-determines the steps to fulfill the agent's goal, `B` could be a perception/action (PA) system that observes and acts on the agent's environment, and `v` could be the visibility of an obstacle. The two interactions could be `A.write(v, yes)` and `B.write(v, no)`, implying that the planner reports that the obstacle can be seen, whereas the PA system reports that the obstacle is hidden. Serializing these interactions causes the final value of `v` to be either `yes` or `no` arbitrarily. However, applying `B`'s interaction and ignoring `A`'s interaction may be a more reasonable, if pessimistic, policy to resolve these interactions. Alternatively, applying `A`'s interaction and ignoring `B`'s interaction may be a reasonable, if optimistic, policy. Another reasonable policy may be assigning weights to the two interactions based on a belief system to produce a multi-modal value for `v`.

In a model of a chemical reaction, `A` could be an acid entity, `B` a catalyst entity, and `v` the volume of a by-product retrieved from the reaction. The two interactions could be `A.write(v, δv`$_A$`)` and `B.write(v, δv`$_B$`)`, where δv_A and δv_B are increases in the value of `v` when `A` and `B` are added. In chemical reactions, it is well-known that adding a catalyst can increase the rate of a reaction tremendously. As a result, the final change in `v` may be more than $\delta v_A + \delta v_B$. Serializing the interactions does not

[*] A detailed analysis of each correctness criterion and policy presented for databases would take up too much time and space. Over 100,000 pages of new material are published every year in databases alone [[7]].

capture the cooperative nature of these interactions. If the interactions must be serialized, then either the model's representation must be augmented with an attribute that keeps track of whether the acid or catalyst has been added previously, or the model must capture the effects of adding a catalyst — an increase in the surface area of the reaction — at a finer level of detail. Alternatively, a special policy must be formulated to increase v appropriately if these concurrent interactions occur.

In the above examples, serializing concurrent interactions produces unintended effects. Isolating them from one another produces effects that are semantically incorrect. Since serialization and other correctness criteria that relax or extend serialization isolate interactions, none of them is a correct policy for resolving them. These interactions are dependent particularly because they are concurrent. Therefore, these interactions require correctness criteria that abandon isolation. The correctness criteria for dependent concurrent interactions are application-specific. Next, with the help of an abstract application, we show how resolving the effects of dependent concurrent interactions by abandoning isolation makes the design of an application complex.

5. Switches — A Simple System

We use a simple system of switches as an abstraction for models with concurrent interactions. We add constraints to the initial model, explaining the effort required to design the corresponding system. Next, we introduce dependent concurrent interactions and show how designing such a simple system becomes complex. We argue that the effects of dependent concurrent interactions must be resolved in an organized manner.

5.1. Unconstrained System

We start with an unconstrained system on which we perform subsequent analyses. Consider the switches S_A, S_1 and S_2 in Figure 1, each with two states: on (or 1) and off (or 0). A client may turn a switch on or off by an interaction (shown by an arrow). The state of the system is an ordered triplet, individual triplet elements being the states of S_A, S_1 and S_2 respectively. In the state transition diagram in Figure 2, an oval is a possible state of the system, a solid arrow is a state transition caused by turning one switch on, and a dashed arrow is a state transition caused by turning one switch off. Transitions that cause the system to begin and end in the same state, for example, turning S_1 off in the state [0 0 0], are not shown in Figure 2 to reduce clutter. Since the switches are independent, all possible states are present in the state diagram.

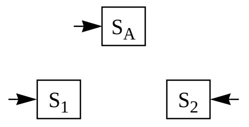

FIGURE 1: Switches

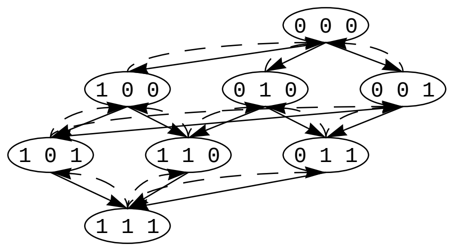

FIGURE 2: State Transition Diagram

5.2. Constrained System

Typically, systems are constrained; their components are related. Accordingly, we add a constraint to our switches: If S_1 and S_2 are both on, then S_A must be on. In other words, $(S_1 = 1) \wedge (S_2 = 1) \Rightarrow (S_A = 1)$. As a result of this constraint, the switches are no longer independent. Figure 3 shows the new version of the switches application with the constraint depicted by arrows between the switches. The arrows merely depict a dependency between switches without outlining the nature of the dependency. The new set of valid states for the system is a subset of the old set of valid states. Figure 4 shows the new set of valid states. The crossed-out state does not exist in the new system.

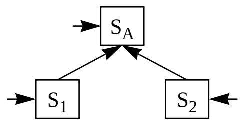

FIGURE 3: Constraints

Usually, constraints reduce the possible states of a system. All transitions going into those states must be redirected elsewhere. The implications of the reduction in the set of valid states on the state transition diagram are shown in Figure 5. The arrows from the states [0 1 0] and [0 0 1] to [0 1 1] have been redirected to [1 1 1] in accordance with the constraint. However, the constraint does not indicate which state to transition from [1 1 1] if only S_A is turned off. In theory, it is possible to transition to any of the seven states (or a hitherto absent state) in such a situation. However, let us abide by the constraint as far as possible. The following are re-statements of the constraint.

S_A	S_1	S_2
0	0	0
0	0	1
0	1	0
~~0~~	~~1~~	~~1~~
1	0	0
1	0	1
1	1	0
1	1	1

FIGURE 4: States

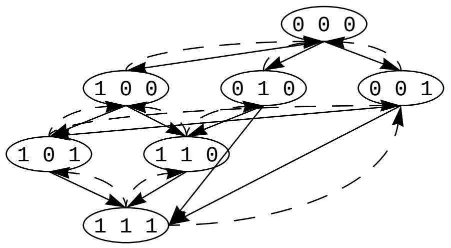

FIGURE 5: Constrained State Transition Diagram

$(S_1 = 1) \wedge (S_2 = 1) \Rightarrow (S_A = 1)$

$\neg((S_1 = 1) \wedge (S_2 = 1)) \vee (S_A = 1)$	[Implication rule]
$\neg(S_1 = 1) \vee \neg(S_2 = 1) \vee (S_A = 1)$	[DeMorgan's laws]
$(S_1 = 0) \vee (S_2 = 0) \vee (S_A = 1)$	[Switch states]
$(S_A = 1) \vee (S_1 = 0) \vee (S_2 = 0)$	[Re-arrangement]
$\neg(S_A = 1) \Rightarrow (S_1 = 0) \vee (S_2 = 0)$	[Implication rule]
$(S_A = 0) \Rightarrow (S_1 = 0) \vee (S_2 = 0)$	[Switch states]

The last statement suggests what to do when S_A is turned off while S_1 and S_2 are on. In order to keep transitions deterministic, we choose [0 0 1] arbitrarily as the state to transition from [1 1 1] in case S_A is turned off, i.e., we turn S_1 off.

State transition diagrams describe a model effectively when sequences of interactions occur. The effects of each interaction are captured by appropriate transitions. Since a state transition diagram can never put the system in an inconsistent state, every interaction can take effect without violating any constraint. Concurrent interactions, whether dependent or not, introduce problems with state transition diagrams, as we show next.

5.3. Dependent Concurrent Interactions

In order to demonstrate the effects of dependent concurrent interactions that cannot be serialized, we add new transitions. Consider the switch system from §5.2, with two concurrent interactions. Let the system be in the state [0 0 1], and let the two interactions be turning S_A off and turning S_1 on. If we serialize them, turning S_A off before turning S_1 on results in the transitions [0 0 1] $\rightarrow$ [0 0 1] $\rightarrow$ [1 1 1], while turning S_1 on before turning S_A off results in the transitions [0 0 1] $\rightarrow$ [1 1 1] $\rightarrow$ s[0 0 1]. The order in which the concurrent interactions are serialized determines the final state of the system. If the final state is immaterial as long as the system stays in a valid state, i.e., a state present in the state transition diagram, then serialization is correct but non-deterministic.

For deterministic behavior, we add other state transitions that capture the effects of concurrent interactions. In Figure 6, we add a transition between [0 0 1] and [0 1 0]. The semantics of this transition could be, for example, that if S_A is turned off and S_1 is turned on *concurrently* in the state [0 0 1], then transition directly to state [0 1 0]. The fact that the interactions were concurrent caused this transition, and the final state of the transition is different from that if the two interactions were serialized.

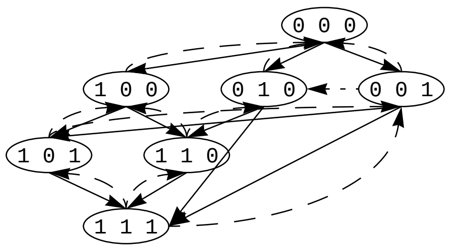

FIGURE 6: Transitions on Concurrent Interactions

5.4. Complexity

We desire systems to behave predictably no matter what interactions occur and how they occur. Accordingly, singly-occurring interactions as well as concurrent interactions must have predictable results. A brute-force approach to resolving the effects of all possible concurrent interactions can be overwhelming. Therefore, a means of encoding dependencies among interactions is necessary.

For the switches system in §5.2, given the six kinds of interactions (turning one of the switches on or off) and the seven different states, an exponential number of transitions are possible on concurrent interactions. In the worst case, the total number of transitions for the switches application is: $(2^{\text{number of interaction types}} - 1) \times \text{number of states} = (2^6 - 1) \times 7 = 447$. This calculation assumes that concurrent interactions of the same kind can be serialized without changing their effect. In other words, concurrent multiple occurrences of the interaction to turn S_1 off, for example, can be serialized. Nevertheless, even in our simple system, the number of transitions that must be considered is large. Applications with more attributes, some non-Boolean, are likely to have many more states than our simple system. Consequently, the number of transitions to be considered can grow further. However, a number of mitigating factors can reduce the number of state transitions for a system. In the switch system, in order to reduce the number of possible transitions, we stated that multiple occurrences of the same interaction can be serialized. Another reasonable assumption is that a switch client will not send concurrent on and off interactions to its switches. This assumption reduces the number of transitions to the product of the number of states and the number of all possible concurrent interactions. The latter number is the sum of concurrent interactions occurring in all combinations of threes, twos and ones. Therefore, the total number of transitions is: $\sum_{i=1}^{3} \binom{3}{i} \times 2^i \times 7 = 182$. This number of transitions shown is an upper-bound, because we assume that no set of concurrent interactions is serializable.

Applications must exhibit predictable behavior when concurrent interactions occur. Serialization is an example of predictability. However, as we have seen in §4, serialization fails to resolve dependent concurrent interactions correctly, because it assumes that the interactions can be isolated. Another example of predictability is commutativity [18], wherein the effects of commutable interactions are the same regardless of the order in which they are applied. Since commutativity also assumes that interactions can be isolated, it cannot resolve the effects of dependent concurrent interactions correctly. When dependent concurrent interactions occur, predictability can be gained by encoding transitions in rigorous formulæ. In such an approach, the behavior of the system when any set of concurrent interactions occur must be encoded *a priori*. Such an encoding is similar to specifying transitions in a state diagram for every possible set of concurrent interactions. As we have shown with our simple switches system, specifying all possible transitions can become a complex task.

We encode semantic information in interactions in our technique for predictable behavior when dependent concurrent interactions occur. Our technique does not isolate interactions, and does not incur the complexity cost of specifying all transitions.

6. A Taxonomy of Interactions

The effects of dependent concurrent interactions are application-specific. Specifying policies for resolving the effects of every set of interactions that may occur concurrently is a complex design task. However, specifying policies for resolving the effects of *classes* of interactions can be less complex. We discuss the properties of a good taxonomy of interactions. MRM designers may classify their interactions into any taxonomy that exhibits these properties. We present and justify one such taxonomy consisting of four classes of interactions. Our taxonomy is based on semantic characteristics of interactions we encountered often in models. Also, we present policies for resolving the effects of classes of concurrent interactions.

6.1. Properties of a Taxonomy of Interactions

A good taxonomy exhibits the properties below [1] [11]:

- *mutually exclusive*: classes do not overlap
- *exhaustive*: classes jointly cover all possible members
- *unambiguous*: classification not dependent on classifier
- *repeatable*: subsequent trials lead to same classification
- *accepted*: logical and intuitive classes
- *useful*: must lead to insights in particular field

MRM designers may choose any taxonomy of interactions as long as it exhibits the above properties. Traditional taxonomies of interactions, for example, reads *versus* writes or serializable *versus* non-serializable, may not exhibit these properties [15].

6.2. Interaction Characteristics and Classes

We show how to classify interactions based on semantic characteristics. We identify four high-level semantic characteristics of interactions. These characteristics are application-independent. The characteristics themselves are well-known; however, using them to classify interactions is novel. We identify four interaction classes from these characteristics of interactions.

Request and Response: Interactions may be distinguished as being requests or responses. Request interactions are concerned with an entity soliciting some behavior from another entity. For example, when an entity queries the status of another entity, the former sends the latter a request interaction. Likewise, if an officer entity orders a soldier entity to fire, the former sends the latter a request interaction. Response interactions are concerned with an entity responding to a request from another or an interaction generated in response to a modeling event. Responses may not be solicited explicitly, i.e., a response may not have a request associated with it. For example, a status update is a response interaction. Likewise, billiard ball entities may send one another response interactions generated because of a collision.

The distinction between request and response interactions is temporal. A request interaction is made regarding a future action. A response interaction is made regarding an action in the past. An interaction may be a request or a response, but not both.

- *Request*: An interaction concerned with eliciting future behavior from an entity.
- *Response*: An interaction concerned with the effects of an action in the past.

Certain and Uncertain: Interactions may or may not have the desired outcomes. Certain interactions have predictable outcomes. For example, when billiard ball entities collide, the outcome of their interaction is predictable because of physical laws. Likewise, when an acid entity is added to an alkali entity, the outcome of their interaction is predictable because of chemical laws. Uncertain interactions are those whose outcomes are not predictable. For example, a request for information may not always be satisfied, or satisfied truthfully. Likewise, a request to perform an action may not be satisfied.

Uncertainty in interactions may be defined along a continuum. For example, interactions may be distinguished on a scale with completely certain interactions at one end and increasingly uncertain interactions further away from that end. In such a case, the uncertainty of an interaction is a measure of its distance from the completely-certain end of the scale. Priorities may be viewed as an example of such a continuum. High-priority interactions always take effect preferentially over lower-priority interactions.

- *Certain*: An interaction whose outcome is predictable.
- *Uncertain*: An interaction whose outcome is unpredictable.

Combining Characteristics: Combining these characteristics gives us four classes of interactions, which we name Type 0, 1, 2 and 3. We list the four classes below along with the conjunction of characteristics that defines each class. Also, we present an example interaction for each class. We depict the four classes in Figure 7.

Type 0:	Response	$\wedge$ Certain	e.g., physical events
Type 1:	Response	$\wedge$ Uncertain	e.g., updates
Type 2:	Request	$\wedge$ Certain	e.g., reads
Type 3:	Request	$\wedge$ Uncertain	e.g., orders

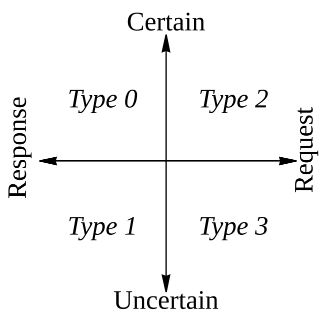

FIGURE 7: Classes of Interactions

6.3. Evaluating the Taxonomy

Our taxonomy of interactions exhibits the properties of a good taxonomy. Our four interaction classes are mutually exclusive since no two of them possess the same conjunction of characteristics. Our taxonomy is exhaustive because the four interaction classes cover all possible combinations of the four interaction characteristics. We believe our taxonomy is unambiguous, repeatable, intuitive and useful. Our characteristics capture semantic information about interactions. An interaction can be classified into our four classes according to *semantic* information, (i.e., its expected effect on its sender and receiver), rather than non-semantic information (e.g., its syntax, the variables it reads or writes, its size, the time taken to transmit it). We assume model designers can identify the semantics of an interaction and determine its characteristics subsequently. Determining the type of an interaction from its characteristics is unambiguous and repeatable. Our classes are logical combinations of orthogonal interaction characteristics. The classes are intuitive because they are derived from well-known characteristics of interactions. All of the interactions we have encountered exhibit these characteristics. Next, we will demonstrate the usefulness of our taxonomy by showing how concurrent interactions can be resolved.

6.4. Resolving Effects of Concurrent Interactions

We show how to resolve the effects of concurrent interactions based on two sets of characteristics of interactions: response *versus* request and certain *versus* uncertain. Independent interactions are those whose concurrent occurrence is indistinguishable from their sequential occurrence. If we can determine that concurrent interactions are independent, then they may be serialized. The following properties enable designers to determine whether concurrent interactions are independent.

Property 1: *If the concurrent occurrence of interactions is indistinguishable from a sequential occurrence, the interactions are independent.*

Argument: Assume the interactions are dependent. Therefore, they are related by either cause-effect or concurrence. If they are related by cause-effect, they cannot occur concurrently, since cause precedes effect. If they are related by concurrence, no sequential occurrence of the interactions can have the same effect as the concurrent occurrence. Since the interactions do not depend on one another by either cause-effect or concurrence, the initial assumption is false. □

Property 2: *If concurrent interactions affect disjoint sets of attributes, they are independent.*

Argument: If concurrent interactions affect disjoint sets of attributes, their effects can be applied sequentially. Therefore, the concurrent occurrence of these interactions is indistinguishable from their sequential occurrence. By Property 1, they are independent. □

If concurrent interactions affect non-disjoint sets of attributes, they *interfere*, but may not be dependent.

Property 3: *Concurrent response and request interactions are independent.*

Argument: Consider the interactions occurring during a time-step $[t_i, t_{i+1}]$ (see Figure 8). Response interactions received during this time-step refer to behavior prior to time t_i. Request interactions received during this time-step refer to behavior after time t_{i+1}. Let there be a time t' such that $t_i < t' < t_{i+1}$. Re-arrange the interactions such that all response interactions occur during the time-step $[t_i, t']$, and all request interactions occur during the time-step $[t', t_{i+1}]$. This re-arrangement does not alter the semantics of any interaction because all of the response interactions continue to refer to behavior prior to time t_i and all of the request interactions continue to refer to behavior after time t_{i+1}. All of the response interactions can occur before all of the request interactions. Therefore, the concurrent occurrence of response and request interactions is indistinguishable from a sequential occurrence, namely, responses before requests. By Property 1, responses and requests are independent. □

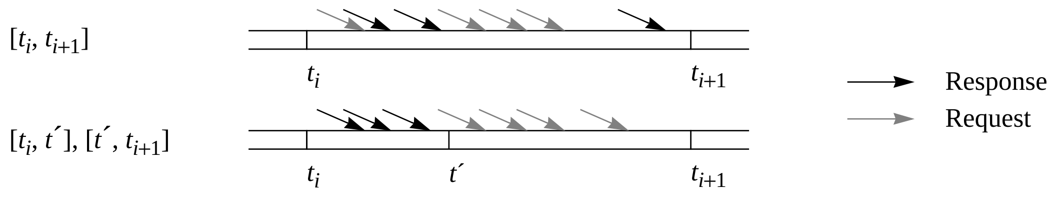

FIGURE 8: Independent Concurrent Response and Request Interactions

When two interactions interfere, but one of them has a certain outcome and the other has an uncertain outcome, then the former takes effect preferentially over the latter. Interactions with certain outcome *must* take effect, whereas interactions with uncertain outcome may be ignored, delayed or permitted to take partial effect. A partial effect for an interaction is the effect of the interaction on some attributes but not others, or a fractional effect of the interaction as opposed to the complete effect. If certainty or uncertainty of interaction outcomes is multi-modal (e.g., as in priorities), then interactions with higher degrees of certainty take effect preferentially over those with lower degrees of certainty.

When two interactions are resolved, either one of them takes effect preferentially over another, or they are combined. In the former case, the preferred interaction retains its type. In the latter case, the resultant interaction has the same type as the original interactions. If interactions of the same type interfere, they must be resolved by application-specific policies. For example, if two Type 0 interactions interfere, then they must be combined by a policy that reflects domain-specific laws. If the interactions cannot be combined, then the model must be re-designed to avoid such paradoxical interactions. When the effect of the combination of some interactions is greater than the combination of effects of the individual interactions, the interactions are *cooperative*. When the effect of the combination of some interactions is less than the combination of effects of the individual interactions, the interactions are *competitive*. If cooperative or competitive effects exist and the original interactions are serialized, new interactions must be added to account for these effects.

6.5. Policies for Resolving Effects of Interactions

We present policies to resolve the effects of dependent concurrent interactions based on the characteristics of interactions. Designers of multiple models may choose from these or similar policies to resolve the effects of dependent concurrent interactions. We present these policies in detail elsewhere [15].

Serializing: If the concurrent effects of some interactions cannot be distinguished from their sequential effects, the interactions are independent (Property 1). Therefore, the effects of independent concurrent interactions may be applied by ordering the interactions and permitting them to take effect one after another.

Ignoring: The effects of some sets of dependent concurrent interactions can be resolved meaningfully by ignoring some of the interactions.

Delaying: The effects of some sets of dependent concurrent interactions can be resolved meaningfully by delaying some of the interactions.

Combining Cooperatively or Competitively: Some dependent concurrent interactions may be resolved by enhancing or diminishing the effects of the individual interactions. The effects of such interactions may be resolved by applying the effects of the individual interactions as well as compensatory interactions that account for cooperative or competitive effects.

7. Constructing an Interaction Resolver

An *Interaction Resolver* (IR) for an entity must resolve the effects of concurrent interactions received by the entity. Resolving interactions involves determining the class of each interaction, determining if interactions of the same type interfere, propagating the effects of interactions and resolving the effects at each attribute using application-specific policies. The IR for an entity may be a single component or a number of components distributed over the attributes in the entity. Conceptually, the distinction is unimportant; during implementation, the distributed view may be more efficient.

7.1. Operation of an IR

The operation of an IR involves encoding and implementing policies for resolving the effects of classes or types of concurrent interactions.

At design time, a designer encodes the type of each interaction and policies for resolving types of concurrent interactions. Encoding the type enables the IR to classify interactions, while encoding the policies enables the IR to resolve interactions. For example, when Type 1 and Type 0 interactions interfere, the former are discarded. The designer must specify a policy for discarding the Type 1 interactions, for example by ignoring or delaying them. The choice of policies may be dynamic, i.e., varying during run-time. However, designers must specify conditions under which the appropriate policy is chosen.

At run time, an entity sends and receives concurrent interactions. An IR for the entity must determine the type of each interaction and group the interactions according to their type. Initially, the effect of each interaction on the set of attributes in all the representations is determined assuming that the interaction occurred in isolation. The semantics of an interaction determine how members in its *affects* set are changed. For each attribute, a list of potential changes caused by the interactions is constructed. Not all of these changes will be applied to the attribute. From its encoded policies, the IR must determine which changes must be applied.

The IR must resolve the changes caused by all interactions by considering the types of the interactions and policies that eliminate conflicts among types of interactions. Based on our classification of interactions, the IR must consider the changes to each attribute in the order

Type 0, 1, 2 and 3. This order preserves dependencies among interactions.

Below, we present an algorithm for an IR. The IR must determine the effects of all concurrent interactions by referring to policies encoded by the designer. In this algorithm, we apply the effects of interactions after all dependent interactions have been resolved.

```
For each time-step
    List L = sort interactions by type
    For each interaction I in L
        Determine effects of I on each attribute
    For each attribute a
        If cooperative/competitive effects exist
            Insert compensatory effects in L
        If Type 0 and 1 interactions interfere
            Discard Type 1 changes
        If Type 2 and 3 interactions interfere
            Discard Type 3 changes
    For each attribute a
        Apply remaining changes
```

When all these changes have been applied, the entity will be consistent. The IR enforces policies meaningful for dependent concurrent interactions. Since the specified policies for dependent concurrent interactions do not isolate the interactions, the effects of these interactions can be resolved meaningfully. Consequently, the entity interacts at multiple representational levels concurrently and consistently. We present an example IR elsewhere [15].

8. Summary

Concurrent interactions may have effects that are dependent on one another. Resolving the effects of such interactions by serializing them is generally incorrect since serialization isolates the interactions. We present some characteristics of interactions — request, response, certain and uncertain — and four classes of interactions based on combinations of these characteristics — Types 0, 1, 2 and 3. The classes distinguish semantic types of interactions encountered commonly in modeling and simulation. Based on these intrinsic characteristics of interactions, we presented policies for resolving the effects of their concurrent occurrence. We showed how to construct an Interaction Resolver (IR) for an entity. An IR encodes policies for resolving the effects of dependent concurrent interactions at run-time. By designing an IR, a designer can ensure that an entity's behavior is meaningful when it interacts concurrently.

9. References

[1] Amoroso, E. D., *Fundamentals of Computer Security Technology*, Prentice Hall PTR, ISBN 0-13-108929-3, 1994.

[2] Badrinath, B. R., Ramamritham, K., *Semantics-Based Concurrency Control: Beyond Commutativity*, ACM Transactions on Database Systems, 17(1), March 1992.

[3] Barghouti, N. S., Kaiser, G. E., *Concurrency Control in Advanced Database Applications*, ACM Computing Surveys, 23(3), September 1991.

[4] Bernstein, P. A., Goodman, N., *Concurrency Control in Distributed Database Systems*, ACM Computing Surveys, 13(2), June 1981.

[5] Bernstein, P. A., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company Inc., ISBN 0-201-10715-5, 1987.

[6] Brahmadathan, K., Ramarao, K. V. S., *On the Management of Long-Living Transactions*, Journal of Systems Software, 11, 1990.

[7] Date, C. J., *An Introduction to Database Systems (Sixth Edition)*, Addison Wesley Publishing Company Inc., ISBN 0-201-54329-X, 1995.

[8] DIS Steering Committee, *The DIS Vision, A Map to the Future of Distributed Simulation*, Comment Draft, October 1993.

[9] Garcia-Molina, H., *Using Semantic Knowledge for Transaction Processing in a Distributed Database*, ACM Transactions on Database Systems, 8(2), June 1983.

[10] Haerder, T., Reuter, A., *Principles of Transaction-Oriented Database Recovery*, ACM Computing Surveys, 15(4), December 1983.

[11] Howard, J. D., *An Analysis of Security Incidents on the Internet 1989-1995*, Ph.D. Dissertation, Engineering and Public Policy, Carnegie Mellon University, 1997.

[12] Korth, H. F., Speegle, G. D., *Formal Model of Correctness without Serializability*, ACM SIGMOD Record, 17(3), September 1988.

[13] Lynch, N. A., *Multilevel atomicity: a new correctness criterion for database concurrency control*, ACM Transactions on Database Systems, 8(4), December 1983.

[14] Munson, J., Dewan, P., *A Concurrency Control Framework for Collaborative Systems*, ACM Conference on Computer Supported Cooperative Work, 1996.

[15] Natrajan, A., *Consistency Maintenance in Concurrent Representations*, Ph.D. Dissertation, Department of Computer Science, University of Virginia, January 2000.

[16] Papadimitriou, C. H., *The Theory of Database Concurrency Control*, Computer Science Press, ISBN 0-88175-027-1, 1986.

[17] Reynolds Jr., P. F., Natrajan, A., Srinivasan, S., *Consistency Maintenance in Multi-Resolution Simulations*, ACM Transactions on Modeling and Computer Simulation, 7(3), July 1997.

[18] Rosser, J. B., *Highlights of the history of the lambda-calculus*, Conference Record of 1982 ACM Symposium on Lisp and Functional Programming, 1992.

[19] Thomasin, A., *Concurrency Control: Methods, Performances and Analysis*, ACM Computing Surveys, 30(1), March 1998.

[20] Weihl, W. E., *Commutativity-Based Concurrency Control for Abstract Data Types*, IEEE Transactions on Computers, 37(12), December 1988.