19 LADA — A Logic For The Analysis Of Distributed Actions

Alan Dix

Abstract

This paper presents a formalism, LADA, aimed especially at the description of systems and situations which arise during the design and analysis of groupware. We are particularly interested in highly distributed systems and so LADA explicitly models entities (people and things) acting at different unconnected locations. It not only describes the behaviour of the computer software, but also the social protocols required for its successful use. Temporal logic formulae which follow the subjective history of people and other entities are used to simplify the expression of some of the complicated properties required of real systems.

19.1 Introduction

Groupware systems are often distributed either over local area networks or over wider distances. In addition, some systems operate on computers which rarely or never directly communicate over a network, instead relying on email or floppy disk transfers. Traditional distributed systems try to minimise the interference between multiple users and even multiple processes, for example, by locking mechanisms. However, *cooperation* between users suggests that this emphasis on transparency is unacceptable and it is replaced by a desire for mutual awareness [9]. Furthermore, traditional distributed systems rarely deal with systems with infrequent communications (although there have been some recent exceptions notable the Coda File System [8]).

Given these extra complications, it is not surprising that groupware systems are difficult to program and even more difficult to debug. In addition, the consequences of crashes are also more serious as they affect many users simultaneously and may reduce confidence in what are often already socially fragile systems [6, 4, Ch. 13].

One would obviously like some form of argument or proof that the central algorithms of these systems behave as required. Unfortunately, such arguments tend to be very complex also as one has to consider all possible combinations of events, including ‘race conditions’ and the possibility of deadlock. It is even difficult to state precisely what the requirements are of a system involving several users distributed over several sites.

One would like to use formal arguments to verify correctness, but most popular formalisms for describing and reasoning about software concentrate on single-threaded systems. Even notations for handling concurrent processes, such as CSP [7], do not deal with issues of distribution and operate at a very low-level.

In addition to these requirements on the computer software, the behaviour of the entire groupware system (human and machine) depends on the participants. Designers make explicit and implicit assumptions about their use of the software. For example, it may be assumed that participants never simultaneously work on the same portion of a document. The designer ought to be able to clearly state what assumptions are being made about these *social protocols* and also investigate the consequences when these assumptions fail.

This paper presents a formalism designed especially for describing actions which happen to people and things at different places. It draws on standard temporal logic, but allows one to trace the subjective experience of each person or thing, rather than a single objective history. The augmented temporal logic is itself built upon a semantic model of partially ordered events. The model and logic are presented at a semi-formal level as there is not room for the full formal semantics and it seemed more valuable to explore a worked example.

In the next section we will consider some of the systems which we would like to describe more formally. This is followed by two sections describing the underlying model and the temporal logic based on it. The major bulk of the paper is an extended example: specifying the Liveware database and giving a sketch proof of one form of observational consistency. Finally, we will return to look at some of the wider issues this example raises about LADA itself and specification in this complex domain.

19.2 Background — cooperating at a distance

If two users are cooperating over the same document (spreadsheet, etc.), but are not working at the same site, they can obviously all have copies on their own local machine. However, if several users simultaneously update copies of the same document, problems arise. Systems for asynchronous group working must either prevent such conflicts by the use of locking or include support to handle the multiple versions which arise.

One example is Liveware [11]. This is a database which is shared by the merging of copies as colleagues meet one another. Changes to the database made by one user gradually spread throughout the community of users as different copies of the database meet and merge with one another. A more complex example of a similar principle is multiple source control [1, 3]. This uses version control mechanisms to allow users to update different copies of the same document at different sites. This causes several ‘streams’ of activity for the same document which can be merged with automated help. In order to avoid sending whole copies of the document, the communication between different sites uses *deltas*, that is differences between versions of the document.

Among commercial systems similar issues arise. Laplink™, the popular PC file transfer tool, allows two file systems to be synchronised based on the date of last update: if the same file exists on both systems, the system overwrites the older version. Of course, if the file has been updated on both file systems, the least recent of the two updates will be lost. Similarly, Lotus Notes™ may have several servers with copies of the same database. If a note is updated on two servers then one update may be lost. This problem is partially ameliorated by the addition of a versioning feature whereby both copies of a note are retained. Although this means that the information is not lost there are potential social conflicts. Imagine that Alison and Brian have each updated the same note on two different servers, but that Alison’s update preceded Brian’s. After the servers synchronise Alison will find that her updates have been superseded by Brian’s. She may wrongfully conclude that Brian actually saw her updates and then rudely ignored them.

This work is being carried out in the context of a project to investigate the use of version support in CSCW systems, and the we believe that the use of appropriate versioning offers the potential to detect such conflicts and offer the participants help in resolving them. However, more complex support system could just mean that problems have simply been put off and will reappear in more complex situations. This emphasises the need for methods to verify that any new systems we propose both work as required and are robust when the social protocols upon which they depend are broken.

Although the principal emphasis in this paper is on asynchronous groupware, the needs in synchronous groupware can be equally pressing. Whenever a system has decentralised control there will be potential for simultaneous conflicting actions. For example, in the Grove synchronous editor, complex algorithms are necessary so that typing can be immediately echoed, but also so that all participants eventually see the same text [5].

19.3 The semantic model

Standard temporal logic and also mathematical time series are based on a linear model of time. A history is modelled as a sequence of states of the world at different times.[1]

$$S_0 \to S_1 \to S_2 \to S_3 \to S_4$$

Temporal logic operators can then be used to talk about such an history, for example, '$\Diamond P$' says 'eventually P is true' or in other words there is some time t for which P holds in the state at that time (S_t). Given such a model we are forced to specify the order in which all actions have happened. For a synchronous system this is reasonable, but for asynchronous groupware this would hide the very issues we want to discuss, namely simultaneous actions in different locations.

We have therefore adopted a model based on a partially ordered set of events. Each event is the occurrence of an action involving one or more entities, which may be human or machine. Events are only ordered where there is some explicit dependency between them, for example if some entity is involved in both events.

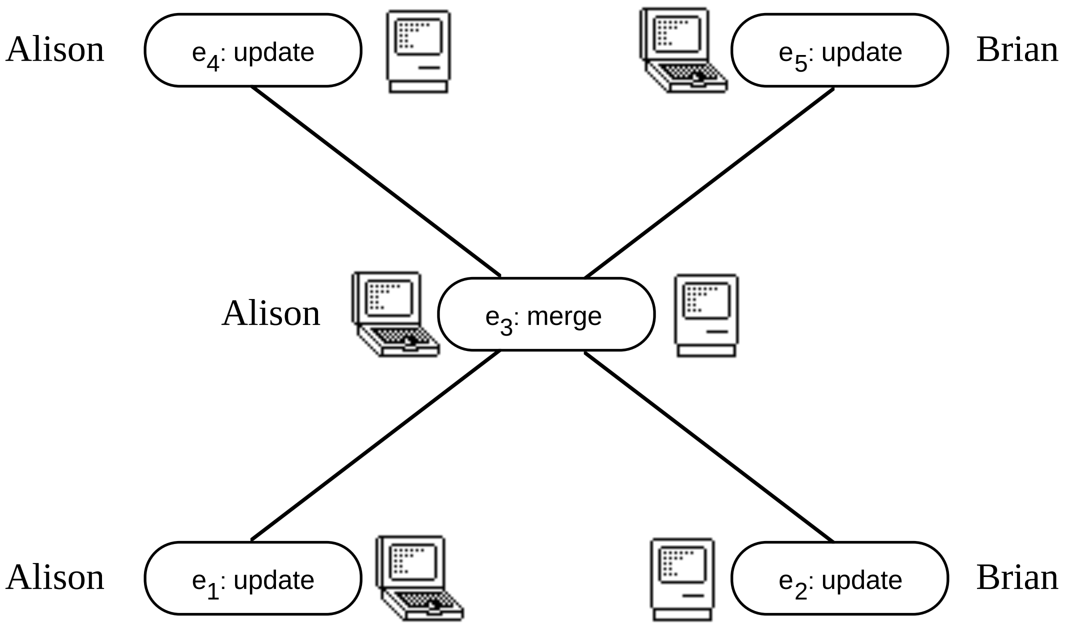

FIGURE 19.1. Event lattice

19.3.1. The lattice of events

In Figure 19.1, Alison and Brian are working on a document together. Initially Alison is working at home updating the document on the mobile computer, whilst Brian is working on the office computer. Event 'e_1' represents Alison's work and 'e_2' represents Brian's work. Later Alison brings the portable into the office and merges the two versions (event 'e_3'). The next day Alison continues to work in the office, but Brian takes the portable to work on the train. Each update the documents during the day on the two machines (the events 'e_4' and 'e_5' respectively).

The diagram shows the partial order between the events so, for example, e_1 happens before e_3 ($e_1 \prec e_3$) and $e_3 \prec e_5$, but we cannot say anything about the order of e_1 and e_2. This is not just a matter of e_1 and e_2 happening at different places — the important point is that they are unsynchronised. Indeed, one could imagine a similar scenario where the two machines are in the same office, but not connected to one another (although then one would expect some events representing communication between Alison and Brian … unless they aren't talking). Furthermore, the communication between the two machines could be via a modem.

19.3.2. Actions and rôles

Let's look more closely at the entities and actions in Figure 19.1. There are four entities: Alison, Brian, the copy of the document on the portable computer and the one on the office desktop computer. There are two types of actions 'update' and 'merge'. The events are instances of these actions and involve a set of participants, who fulfil rôles for the action. The update action requires a person who performs the update and a document which is changed. The merge action requires two documents to merge, and also a person who initiates and controls the merge. There will typically also be some other information about the event, for example, describing the nature of the update. This has been omitted in Figure 19.1, but is also important. Figure 19.2 shows these relationships in a tabular fashion and includes a description of the actual updates.

	Action	Rôle	Entity	Details
e_1	**update**	*person*	Alison	insert "in the town" after line 2
		document	portable	
e_2	**update**	*person*	Brian	delete line 3
		document	desktop	
e_3	**merge**	*initiator*	Alison	
		doc 1	portable	
		doc 2	desktop	
e_4	**update**	*person*	Alison	change line 2 to "had a shop"
		document	desktop	
e_5	**update**	*person*	Brian	change line 3 to "in the country"
		document	portable	

FIGURE 19.2. Events and their participants

19.3.3. Entity timelines

Finally, we need to add some meaning to the actions. For each action we need to say how it changes the state of its participants. Also for each entity we need to keep track of its state at any time. As the model is distributed we can not ask, for example, about the state of the portable computer at event e_2. However, it is important that the state of the portable is the same at the start of e_3 as it was at the end of e_1. That is, for each entity there is an alternating sequence of states and events which represent its own particular timeline. For example, Figure 19.3 shows the states of the document on the two computers. Depending on our application, we may not want to talk about all of the state of an entity, and for some entities we may not record any state at all. For such entities, the timeline merely records the history of activity of the entity. For example, we can represent the history of Alison and Brian as:

$$Anne_0 \rightarrow e_1 \rightarrow Anne_1 \rightarrow e_3 \rightarrow Anne_2 \rightarrow e_4 \rightarrow Anne_3$$

$$Brian_0 \rightarrow e_2 \rightarrow Brian_1 \rightarrow e_5 \rightarrow Brian_2$$

The symbols $Anne_0$, $Brian_1$ etc. are there to record the fact that people do have some state (memory etc.) but that we are not going to expand upon it.

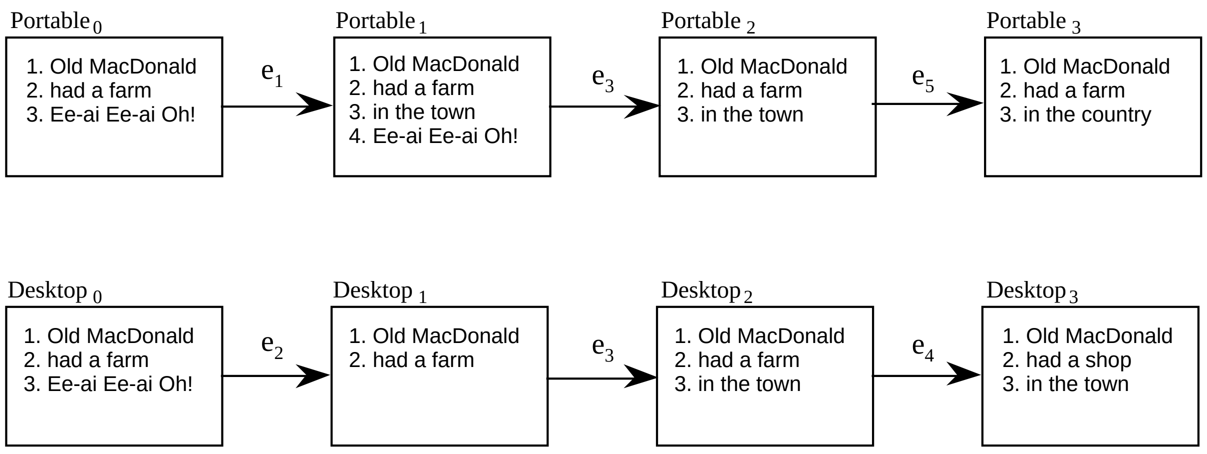

FIGURE 19.3. Document timelines

We are modelling physical entities, such as people, computers and individual copies of a document thus the timeline of any individual entity will be linear — an entity cannot be unsynchronised with itself! This is in contrast to an information entity like the idea of the document “LADA paper”. An information entity can have several different versions in different places, and even have several versions on the same device. We of course want to reason about information entities, such as the evolution of a shared document, but to achieve this we need to talk first about the life histories of the individual versions and the way they interact.

As well as this linearity constraint, we could also demand that events are only ordered if some entity has participated in both event, or if there is some intervening sequence of events with this property. To be a little more precise, we can say that an event e_1 is just before another event e_2 if there is no intervening event. We will write this $e_1 \rightarrow e_2$, and define it formally by:

$$e_1 \rightarrow e_2 \quad \Leftrightarrow \quad (e_1 \prec e_2) \wedge \neg\exists e\ st.(e_1 \prec e \wedge e \prec e_2)$$

The *completeness* condition would then be:

$$e_1 \rightarrow e_2 \quad \Rightarrow \quad \exists \text{ an entity } x \text{ st. } (x \text{ participates in } e_1 \wedge x \text{ participates in } e_2)$$

Although the linearity condition is a necessary sanity condition for concrete entities, there may be situations where completeness is not required. For example, the ordering of some events may be determined by another observer who is not explicitly included in the model. Thus the completeness condition is left as an optional part of the model which can be imposed if required.

19.3.4. Alternative histories

The event lattice in Figure 19.1 shows one possible history of the world. There are lot of other things which could have happened, Alison might have taken the portable home after merging the machines, Brian might not have performed any updates and simply waited to see

Alison's version. The full formal model underlying LADA talks about the set of all possible histories. Given a description of the behaviour of the computer systems and the people involved, only some of these worlds can happen. To prove that some requirement holds we need to show that all possible histories which can happen according to these behaviours also satisfy the required property.

More formally: let *w* be a history of the world — that is, *w* is an event lattice like the one in Figure 19.1, with associated states of entities etc.. We then denote the expected behaviours of the computer systems and people by S(w) and P(w) respectively and the requirement on the system as a whole by R(w). The requirement holds if:

$$\forall \text{ world histories } w \ \left(S(w) \wedge P(w) \Rightarrow R(w)\right)$$

The requirements and behaviour can be specified using formulae like that of the completeness condition, but temporal operators act as 'sugar' so that one can avoid talking about the event lattice directly.

19.4 Temporal operators

Standard temporal logic has two principle operators, the diamond operator – 'eventually', that we have already seen and the box operator – 'always'. We could express the old adage that every cloud has a silver lining something like:

$$\Box \text{ times are bad} \Longrightarrow \Diamond \text{ things get good}$$

Which literally translated says "it is always true that if times are bad then it will eventually be true that things get good".

As eventually can be a long way off, various additional operators are also used. Two of these are 'until' (**U**) and 'before' (**B**). The formula 'p **U** q' says that p will be true of all states until there is a state where q is true. Similarly 'p **B** q' says that before the next state where q is true there will be a state where p is true. The before operator can be defined in terms of until, and visa versa:

$$\mathrm{p}\ B\ \mathrm{q} \Leftrightarrow \neg\left((\neg\mathrm{p})\ U\ \mathrm{q}\right)$$

We cannot use these operators directly to distributed actions as there is no simple time ordering. However, each entity has a time ordering, so these operators can be used along a particular entity's timeline. The entity concerned is added as a subscript. For example, we might want to say that whenever a user types the 'send' command the computer will eventually fax a document. This can be written:

$$\Box_{user} \text{ types}(user, computer, \text{"send"}) \Longrightarrow \Diamond_{computer} \text{ faxes}(computer, \text{the file})$$

Notice how this formula starts off by looking at the user's timeline and then in a sense branches off along the computers timeline. In fact, this is quite a complicated behaviour and many properties can be represented with respect to a single timeline. For example, we may want to say that if a user has been updating a document on one computer and then wants to use a different computer, the user ought to ensure that the two computers have performed a merge before starting work with the second. (Notice how difficult it is to express these requirements — this is precisely why a formalism is needed.) We can express this requirement as:

$$\Box_u \text{ update}(u, comp_1) \Longrightarrow \left(\text{merge}(u, comp_1, comp_2)\ B_u \text{ update}(u, comp_2)\right)$$

This is an example of a social protocol which could be used together with the specification of the system to reason about correctness.

19.5 Example — Liveware

We will now look at the Liveware system in detail and show how one can describe the required properties and go about proving them.

A Liveware database is a collection of individual records which are the units used for update and merging. Each record has a unique identifier generated using the time and user who first created it. So, when two databases are merged the system is able to match records and update the older one to be the same as the newer. The result is that as a group of users meet one another and exchange databases (usually using copies on floppy disk) the data becomes more 'up to date' and information entered by one user gradually permeates throughout the user population. Liveware avoids the concurrent update problem by only allowing a record to be updated by its creator.

19.5.1. Entity states

To describe the system we need to consider two kinds of entities, copies of the Liveware database and users:

entities: Db, User

At any moment, the database state (Db) has a set of active record identifiers. For each identifier, there is an associated value and also a timestamp of when it was last updated:

state: Db =

records:	$\mathbb{P}$ Id
vals:	records $\rightarrow$ ValD
stamp:	records $\rightarrow$ Time

The identifiers are, as was mentioned, constructed using the creation time and the user who created/owns the record. Also the value set will be assumed to include a special value 'DEL', which will be used to record deleted records:

$$\begin{aligned} \text{Id} &= \text{Time} \times \text{UserName} \\ \text{ValD} &= \text{Val} \cup \{\text{ DEL }\} \end{aligned}$$

Given an element id from Id, we will refer to its components as id.t and id.u respectively.

Given any database, we can determine the last time it was updated:

$$\text{lasttime} = \max\left(\{\text{id.t} \mid \text{id} \in \text{records}\} \cup \textbf{range}\ \text{stamp}\right)$$

We would also expect that records cannot be updated before they are created:

$$\forall \text{id} \in \text{records} \quad \text{id.t} \leq \text{stamp(id)}$$

However, although this sounds like a sensible condition, it depends on the way timestamps are generated, in a distributed environment we cannot necessarily assume a global clock. In fact, we will assume a clock that for each action supplies the time as a parameter 'time?'. This will be assumed to satisfy the locally monotonic property:

time 1.

$$\forall e_1, e_2 : \text{Events} \quad e_1 \prec e_2 \Longrightarrow e_1.\text{time?} < e_2.\text{time?}$$

If this is true, then we can assume for any entity ‘ent’:

time 2.

$$\Box_{ent}\ t = time? \Longrightarrow \Box_{ent}\ t < time?$$

That is any entity will see the parameter ‘time?’ increase.

The ‘state’ of users will be considered to consist of their name only:

state: User =
name: Name

Unlike the database state, this will not change. In the following description, we will require the user’s name for most actions. In reality this is likely to be obtained once per session when the database is opened, but for simplicity we will ignore this additional system behaviour.

19.5.2. Action descriptions

We will consider five actions:

create:	add a new record
delete:	destroy a record
change:	change an existing record
look:	examine an existing record
merge:	merge two copies of the database

The effect of an action is a state transformation of the entities involved in the action. This could be described using any suitable notation, for example, Z schemas with suitable additional semantics. In the descriptions of the actions, some Z conventions will be used. For each entity involved with an action, its state variables before the action will be available in the action’s precondition and both these and primed versions available for its post condition. In addition, extra parameters will be included for each action. These will be decorated by either ‘?’ or ‘!’. The former represent parameters which are inputs to the action and can be used in the actions precondition. The latter are regarded as outputs. In reality, the inputs would often originate with one or other of the participants of an action (in this example mainly the user), but the source of these additional parameters is deliberately left underspecified. Of course, this leaves open the possibility of refining the specification to determine some of the input parameters later during system design.

We now consider the effect of each action in turn, describing the entities involved in the action, the additional parameters and the pre and post conditions of the action. First of all record creation:

action: create **roles** : User, Db
params: time?: Time, val?: Val, id!: Id

pre: $(time?, name) \notin records$

post: id! = (time?, name)
records’ = $records \cup \{ id! \}$
vals’ = $vals \oplus \{ id! \rightarrow val? \}$
stamp’ = $stamp \oplus \{ id! \rightarrow time? \}$

This action is only valid when the new record identifier which will be generated by the action is not already in the database. However, if the database never contains anything that is dated in the future the precondition will always be true. To be precise, the condition on the database is:

$$\square_{db}\ \text{lasttime} < \text{time?}$$

In fact, we will see that for each action we define:

$$\square_{db}\ \text{action(db)} \wedge \text{lasttime} < \text{time?} \Longrightarrow \text{lasttime}' \leq \text{time?}$$

This together with the local monotonicity assumption on time will mean that if the database starts in a well timed conditions, it will remain so.

The change action is similar except here the record identifier is required as a parameter:

action: change **roles** : User, Db
params: time?: Time, val?: Val, id?: Id

pre: $\text{id} \in \text{records} \ \wedge\ \text{id.u} = \text{name} \ \wedge\ \text{vals(id)} \neq \text{DEL}$

post:
$$\begin{aligned} \text{records}' &= \text{records} \\ \text{vals}' &= \text{vals} \oplus \{\, \text{id} \to \text{val?} \,\} \\ \text{stamp}' &= \text{stamp} \oplus \{\, \text{id} \to \text{time?} \,\} \end{aligned}$$

Here the precondition is important, it demands that a valid identifier is given. To be valid, it must both be an identifier that is known to the system but has not been deleted. Liveware records that a record has been deleted by storing 'DEL' in its value. In addition, the record must belong to the user.

The delete operation is identical to change except the value is set to 'DEL':

action: delete **roles** : User, Db
params: time?: Time, id?: Id

pre: $\text{id} \in \text{records} \ \wedge\ \text{id.u} = \text{name} \ \wedge\ \text{vals(id)} \neq \text{DEL}$

post:
$$\begin{aligned} \text{records}' &= \text{records} \\ \text{vals}' &= \text{vals} \oplus \{\, \text{id} \to \text{DEL} \,\} \\ \text{stamp}' &= \text{stamp} \oplus \{\, \text{id} \to \text{time?} \,\} \end{aligned}$$

The look action is even simpler, it performs no state changes at all:

action: look **roles** : User, ≡Db
params: time?: Time, id?: Id, val!

pre: $\text{id} \in \text{records} \ \wedge\ \text{vals(id)} \neq \text{DEL}$

post: $\text{val!} = \text{vals(id)}$

The notation '≡Db' comes from Z and means 'no change in Db'.

Notice also that the precondition is weaker: users can look at any record, not just their own. Of course, a more sophisticated treatment could consider access control issues, but this will be ignored here.

Finally, we consider the merge operation. This involves two databases, they start possibly different and end up the same. Each record in the final state is the most up to date from the two sources:

action: merge **roles** : $\mathbb{P}$ User, Db_1, Db_2

post: $\text{records}_1' = \text{records}_2' = \text{records}_1 \cup \text{records}_2$

$\forall\ \text{id} \in \text{records}_1'$ *st.* $\text{stamp}_1(\text{id}) < \text{stamp}_2(\text{id})$

$$\begin{aligned} \text{vals}_1'(\text{id}) &= \text{vals}_2'(\text{id}) = \text{vals}_2'(\text{id}) \\ \text{stamp}_1'(\text{id}) &= \text{stamp}_2'(\text{id}) = \text{stamp}_2'(\text{id}) \end{aligned}$$

$\forall\ \text{id} \in \text{records}_1'$ *st.* $\text{stamp}_1(\text{id}) \geq \text{stamp}_2(\text{id})$

$$\begin{aligned} \text{vals}_1'(\text{id}) &= \text{vals}_2'(\text{id}) = \text{vals}_1'(\text{id}) \\ \text{stamp}_1'(\text{id}) &= \text{stamp}_2'(\text{id}) = \text{stamp}_1'(\text{id}) \end{aligned}$$

Notice that there are no preconditions on merge, neither does it actually need any information from the users. A set of users is given as more than one user may be present (and often will be) when databases are merged. However, not all owners of records need be present; the records from all users are synchronised. This does not mean that a user is actually updating another user's records since whichever value is in the final databases will have been set by that user anyhow.

19.5.3. Generic Actions

In order to simplify the requirements on a system, it is useful to classify the actions into groups of generic actions, giving a class structure to them. Three classes will be used:

update(user,db,id)	≡	create or change or delete
sees(user,db,id)	≡	update or look
interact(user,db)	≡	sees or merge

The last class is intended to capture when the user interacts with a database in any way. As a merge involves two databases and possibly several users, it may be part of the generic action in several ways.

19.5.4. Observational consistency

Now we can capture some of the Liveware properties described in [10]. They note that the collection of Liveware databases at any time are *not* necessarily consistent with one another. However, even though they are not *globally* consistent, they are *observationally* consistent for any particular user. That is although there may be inconsistencies, no one will ever notice!

This statement has several ramifications and one can address it at various levels. However, we will just look at one: if a user sees the same record take on different values, it must always get more up to date.

<u>obs. 1.</u>

$$\Box_{\text{user}}\ \text{sees}(\text{user}, \text{db}_1, \text{id}) \wedge \text{stamp}_1{}'(\text{id}) = \text{t} \implies$$
$$\Box_{\text{user}}\ \big(\text{sees}(\text{user}, \text{db}_2, \text{id}) \Rightarrow \text{stamp}_2{}'(\text{id}) \geq \text{t}\big)$$

With no social protocol this property cannot be guaranteed. Consider for example the following event history for the user, where db_1 and db_2 are separate copies of the database:

$$\text{update}(\text{user}, \text{db}_1, \text{id}, \text{val}_1) \;\rightarrow\; \text{update}(\text{user}, \text{db}_2, \text{id}, \text{val}_2) \;\rightarrow\; \text{look}(\text{user}, \text{db}_1, \text{id})$$

Clearly, the final timestamp of id will be older than the timestamp in the previous event, violating the required condition. We therefore need a social protocol to make the system work. We use similar conditions to the example formula earlier:

<u>social 1.</u>

$$\Box_{\text{user}}\ \text{merge}(\text{user}, \text{db}_1, \text{db}_2) \implies$$
$$\left(\begin{array}{l} \text{sees}(\text{user}, \text{db}_3) \Rightarrow (\text{db}_1 = \text{db}_3 \vee \text{db}_1 = \text{db}_3) \\ \qquad\qquad \mathcal{U}_{\text{user}} \quad \text{merge}(\text{user}, \text{db}_1, \text{db}_4) \vee \text{merge}(\text{user}, \text{db}_2, \text{db}_4) \end{array}\right)$$

social 2.

$$\Box_{\text{user}}\ \text{sees}(\text{user},\text{db}_1) \Longrightarrow$$
$$\big((\text{sees}(\text{user},\text{db}_3) \Rightarrow \text{db}_1 = \text{db}_3)\ \mathcal{U}_{\text{user}}\ \text{merge}(\text{user},\text{db}_1,\text{db}_2)\big)$$

The first condition says that between merges the user only interacts with one of the databases which were merged. the second says that when the user has chosen to interact with a database that is the only one which is used until the next merge.

19.5.5. Sketch proof of observational consistency

The proof is in three main steps. First one proves the equivalent property for a single database following its timeline. That is:

proof. 1.

$$\Box_{\text{db}}\ \text{id} \in \text{records}' \wedge \text{stamp}'(\text{id}) = t \Longrightarrow \Box_{\text{db}}\ (\text{id} \in \text{records}' \wedge \text{stamp}'(\text{id}) \geq t)$$

This is proved by induction and case analysis over the actions which a database can be involved in. Indeed, it is trivial to see that no action may reduce a records timestamp.

The second stage is to transfer this result to a user looking at a single database. That is we seek to prove:

proof. 2.

$$\Box_{\text{user}}\ \text{sees}(\text{user},\text{db},\text{id}) \wedge \text{stamp}'(\text{id}) = t \Longrightarrow$$
$$\Box_{\text{user}}\ (\text{sees}(\text{user},\text{db},\text{id}) \Rightarrow \text{stamp}'(\text{id}) \geq t)$$

Although this involves only the user and one database, it s no longer a linear temporal logic proof. Both the user and the database may have interactions which do not involve one another. In other words, we need to prove temporal properties of entities which take different time paths.[2] For the first time, we really need the additional expressive power of a non-linear logic and can use a general proof rule. Let a and b be any two entities, and e and f two action types which involve them both:

infer 1.

from $\Box_a\ P \Longrightarrow \Box_a\ Q$

infer $\Box_b\ e(a,b) \wedge P \Longrightarrow \Box_b\ f(a,b) \Rightarrow Q$

We simply set P to be $(\text{id} \in \text{records}' \wedge \text{stamp}'(\text{id}) = t)$ and Q to be $(\text{id} \in \text{records}' \wedge \text{stamp}'(\text{id}) \geq t)$. The first part of our proof (*proof 1.*) has therefore fulfilled the condition of the proof rule. Examining each specific action which comprises the generic action ‘sees(user,db.id)’, we find that all imply that id is in records’. Hence letting both events e and f be ‘sees(user,db.id)’ we obtain equation *proof 2.*

So far we are still only considering a user’s interactions with one database. To prove the full observational consistency property, we need to use the social protocol. This is now a linear temporal logic step and just involves induction and case analysis. The important fact is that the social protocol ensures that the user’s interactions are always of the form:

$$\text{sees}(\text{user},\text{db}_1) \rightarrowtail \ldots \rightarrowtail \text{sees}(\text{user},\text{db}_1) \rightarrowtail \text{merge}(\text{user},\text{db}_1,\text{db}_2) \rightarrowtail \text{sees}(\text{user},\text{db}_2)$$
$$\rightarrowtail \ldots \rightarrowtail \text{sees}(\text{user},\text{db}_2) \rightarrowtail \text{merge}(\text{user},\text{db}_2,\text{db}_3) \rightarrowtail \text{sees}(\text{user},\text{db}_3) \rightarrowtail \ldots$$

The single database version effectively proves the property between merges, but the fact that databases are identical after merges then allows an inductive proof of *obs. 1.*

19.5.6. Other forms of observational consistency

The proof of simple observational consistency only depended on the system behaviour and the social protocol of the particular user involved. That is, so long as a user obeys the required social protocol any individual record will always get more up to date.

More complex observational properties should look at groups of records. For example, we might like it to be true that two users could agree about their observed history of records. That is, given two records (with ids i and j), and two users (say Alison and Brian), if Alison sees record i updated before record j, we might like Brian to see the same relative ordering. We can define this ordering precisely. We say that Alison sees record i at time t before she sees record j at time 2 (denoted $i@t <_A j@s$) when the following holds:

$$i@t <_A j@s \;\equiv\; \Diamond_A \; \text{sees}(A, db) \wedge \text{stamp}'(i) \geq t \wedge (\, j \notin \text{records}' \vee \text{stamp}'(j) < s\,)$$

This is rather strong as it says that the updates cannot even be first seen at the same time. The weaker equivalent allows this:

$$i@t \leq_A j@s \;\equiv\; \neg\big(j@s <_A i@t\big)$$

Note that although these are written as partial orders, they will only be guaranteed to be so if Alison obeys her social protocol.

The joint consistency property can then be stated:

<u>*obs. 2.* – false</u>

$$i@t <_A j@s \quad\Rightarrow\quad i@t \leq_B j@s$$

Unfortunately, this consistency condition (which is what one would expect of a synchronous database) does *not* hold of Liveware. For example, imagine that record i belonged to Alison and j to Brian. They start with identical copies of the database, then in different locations Alison updates record i and Brian updates j. Clearly Alison will see the update to i before j and visa versa for Brian.

Although this stronger form of the consistency property does not hold for any pair of records, it does hold if the records belong to single user. That is:

<u>*obs. 3.*</u>

$$i.u = j.u \;\wedge\; i@t <_A j@s \quad\Rightarrow\quad i@t \leq_B j@s$$

Proving this requires that all three parties (Alison, Brian and the user to whom the records belong) all obey the social protocol. These three need not be distinct leading to two special cases.

- If Alison and Brian both maintain the social protocol, then each will see the other’s records in a consistent manner.
- If a single user (say Alison) maintains the protocol, then she will always see her own records as consistent with one another.

Note that in both these cases, we put no stipulations on the other users. This is very important as in a large informal group, we cannot guarantee that everyone else will always fulfil their commitments.

19.6 Lessons about the Logic

Having seen a partially worked through example, we can look again at the underlying formalism and see what can and cannot be done within the logical formulae.

19.6.1. Additional temporal operators

Several of the steps in the proof of simple observational consistency involved reasoning about a single entity's timeline. The expresion of properties and proofs are then pretty much standard linear temporal logic.

Inductive proofs of 'always' properties is made easier by using the temporal logic 'next state' operator ($\bigcirc$). As with the other operators, this is decorated with the name of the entity which is being traced. For example, the social protocol can be re-expressed as:

social 1a.

$$\Box_{\mathrm{user}}\ \mathrm{merge}(\mathrm{user}, \mathrm{db}_1, \mathrm{db}_2) \Rightarrow \bigcirc_{\mathrm{user}} \left(\mathrm{interact}(\mathrm{user}, \mathrm{db}_1) \vee \mathrm{interact}(\mathrm{user}, \mathrm{db}_2) \right)$$

social 2a.

$$\Box_{\mathrm{user}}\ \mathrm{sees}(\mathrm{user}, \mathrm{db}_1) \Rightarrow \bigcirc_{\mathrm{user}} \left(\mathrm{interact}(\mathrm{user}, \mathrm{db}_1) \right)$$

However, this formulation is only correct so long as the user only engages in interactions with databases. We might have an additional action for users 'drink_cup_of_tea'. If the user, say Brian, were to follow the above formulation of the social protocol it would imply that once an interaction with a Liveware database had taken place, Brian would have to spend the rest of his life with Liveware and *never* drink another cup of tea!

What we really want to say is "in the next state where there is database interaction...". That is, we want to have temporal operators which only look at actions of a particular type. We can denote this by putting the relevant action condition after the entity name in the temporal operator's decoration. For example:

social 2b.

$$\Box_{\mathrm{user}}\ \mathrm{sees}(\mathrm{user}, \mathrm{db}_1) \Rightarrow \bigcirc_{\mathrm{user:interact}} \left(\mathrm{interact}(\mathrm{user}, \mathrm{db}_1) \right)$$

Similar annotations can be used with the 'always' and 'eventually' operators, but are redundant as the same effect can be achieved in the predicate part:

$$\Box_{\mathrm{ent:action}}\ P \equiv \Box_{\mathrm{ent}}\ \mathrm{action} \Rightarrow P$$

The following two proof rules then allow easy inductive proofs:

infer 2.

from $\Box_{\mathrm{ent:action}} P$

infer $\Box_{\mathrm{ent}} \bigcirc_{\mathrm{ent:action}} P$

infer 3.

from initially P **and** $\Box_{\mathrm{ent:action}} \left(P \Rightarrow \bigcirc_{\mathrm{ent:action}} P \right)$

infer $\Box_{\mathrm{ent:action}} P$

The first of these is a weakening rule. The second, *infer 3.*, requires information about the initial states of entities, which was mentioned once or twice loosely during the previous proofs, The second part the condition (the inductive part) will usually be discovered by recourse to case analysis on the different types of possible action.

19.6.2. Resynchronisation

Not every property or proof step can use purely linear temporal reasoning. If they could there would be no need for a new formalism! In the proof of simple observational consistency (*obs. 1.*), we only needed one proof step involving the intertwining of timelines which we could achieve using the inference rule *infer 1*. However, the more complicated observational consistency property (*obs. 3.*) talks about properties of two timelines (in fact three including the record's owner). Furthermore, the Multiple Source Control system [1] uses asynchronous message transfer as well as direct merging and will thus have even more non-linear features.

The logic formulae we have seen so far make it easy to branch onto different entities timelines. Take for example:

$$\square_{C}\ \mathrm{act}_1(\mathrm{C,A,B}) \ \Rightarrow\ \big(\mathrm{P(A)}\ \mathcal{U}_{\mathrm{A}}\ \mathrm{act}_2(\mathrm{A,B})\big) \wedge \big(\mathrm{Q(B)}\ \mathcal{U}_{\mathrm{B}}\ \mathrm{act}_2(\mathrm{A,B})\big)$$

This says that it is always true along C's timeline that if C engages in the action act_1 then along A's timeline P will hold until A engages in the action act_2 with B and similarly along B's timeline Q would hold until B engages in act_2 with A. We could instantiate this with the following entities an actions:

A	–	Alison
B	–	Brian
C	–	Cupid
act_1(C,A,B)	–	Cupid fires his arrows of passion at Alison and Brian
P(A)	–	Alison is unhappy
Q(B)	–	Brian is sorrowful
act_2(A,B)	–	Alison and Brian marry

The formula would then read:

"Whenever Cupid fires his arrows of passion upon Alison and Brian, Alison will be forever unhappy until she marries Brian and likewise Brian will be in perpetual sorrow until he marries Alison."

However, the formula does not specify whether the two marriages spoken of are the same event. We could imagine a scenario whereby Alison and Brian are both unhappy and then they marry. Brian is now delighted, but Alison stays unhappy and sadly they divorce (presumably casting Brian once more into the depths of sorrow). Only then does Alison realise the depth of her love for Brian and she remains unhappy until years later they meet again, refresh their romance and remarry.

It makes a good story, but was not what the original formula was meant to express. It allowed one to talk about the diverging timelines of Alison and Brian from one event (the shooting of the arrows), but did not demand that the timelines resynchronise on a single event (marriage). In this example, we could force the required interpretation by adding the requirement to P and Q that no marriage takes place. However, this is both inelegant and will not cope with slightly more complex situations. To allow more general resynchronisation we can label the events:

$$\square_{C}\ \mathrm{act}_1(\mathrm{C,A,B}) \ \Rightarrow$$
$$\exists \mathrm{e{:}Event}\ \ st.\ \big(\mathrm{P(A)}\ \mathcal{U}_{\mathrm{A}}\ \mathrm{e{:}act}_2(\mathrm{A,B})\big) \wedge \big(\mathrm{Q(B)}\ \mathcal{U}_{\mathrm{B}}\ \mathrm{e{:}act}_2(\mathrm{A,B})\big)$$

This form of labelling makes the expressive power of the temporal formulae more symmetric (although not completely) at the expense of more complex logic.

Happily, most properties seem to concern relatively little resynchronisation and it is likely that most proofs can be split into mainly linear parts with only occasional non-linear parts. This was certainly the case with the observational consistency proof.

19.6.3. When logic fails

Although the temporal formulae make it easier to define and reason about properties, they are not complete with respect to the underlying semantic model. Occasionally, properties may need to be stated using the model itself (as was the case with the first time property *time 1*.). Even where the required properties can be stated using the temporal formulae, proofs may have to drop into the semantic domain. This is not a fundamentally difficult process as the formulae can be easily translated into their semantic equivalents to carry out proofs there.

If some constructs seem to recur they may become candidates for later inclusion in the logical level of LADA (as is the case with labelled events), but I am not convinced of the utility of aiming at completeness at the logical level when the semantic level is not unduly complex.

If such a dual level approach seems inelegant to the purist it should be noted that temporal formulae are simply predicates about event lattices. If required they can thus be embedded into properties at the semantic level, giving rise to a ‘wide spectrum’ notation. For example, we could say:

$$\Box_a\ \mathrm{act}_1(a) \;\Rightarrow\; \exists e{:}\mathrm{Event}\quad \mathit{st.}\quad \left(\Diamond_a\ \ e{:}\mathrm{act}_2(a)\right) \wedge \left(\forall f \succ e\ \ P(f)\right)$$

Some care has to be taken over the precise semantics of embedded quantifiers over events.

19.6.4. Not just atomic events

Whereas most specification notations for interfaces are targeted at continually synchronised systems, LADA is explicitly aimed at the opposite extreme. The entities meet and synchronise at discrete and atomic events. However, there are situations between these two extremes. For example, we may want to describe systems where participants both work separately, but also have protracted periods of synchronised work. During these periods it is reasonable that participants will join and leave the synchronous group. This is not allowed in the current formalism as this would lead to violations of the partial order between events.

Another way in which the atomic nature of events is insufficient to describe interface phenomena is when there are periods of continuous interaction between events. This *interstitial* (that is between actions) *behaviour* become important when studying more detailed interaction. Although it can be ignored then for the highly distributed systems which are the main target of LADA it suggests that appropriate extensions would be valuable for certain target domains. This would bring together the work described in this paper with a longer running study of status/event analysis (e.g., [2] and [4, Ch. 9]).

19.7 Discussion

It is difficult enough to specify and reason about single user interaction – the movement into groupware with the attendant problems of distribution is daunting. We have seen how LADA is designed to address such systems. It has two major features:

- a partially ordered event lattice to express issues of distribution.
- extended temporal logic formulae following individual timelines.

The first of these is not uncommon, although it is important to remember that the use here is to represent events which are totally unsynchronised, rather than simply things which may happen concurrently. The emphasis here is on distribution not concurrency.

The use of temporal formulae is I believe novel and adds considerably to the ease of use and expressiveness of the resulting formalism. This was evident in the worked example where comparatively complex properties of multiple agents and objects were expressed succinctly and proofs of properties nicely factored into linear and non-linear parts.

Acknowledgements

This work was funded by a SERC Advanced Fellowship B/89/ITA/220 and by SERC Grant GR/J08560, “Version Management and Access Control Models for Cooperative Systems”.

References

[1] A. J. Dix and V. C. Miles, *Version Control for Asynchronous Group Work,* YCS 181, University of York, 1992.

[2] A. J. Dix, “Beyond the interface”, pp. 171-190 in *Engineering for Human–Computer Interaction: Proceedings of IFIP TC2/G2.7 Working Conference,* Ellivouri, Finland, J. Laron and C. Unger editors, North-Holland, 1992.

[3] A. J. Dix & R. Beale, *Information Requirements of Distributed Workers,* University of York, 1992.

[4] A. Dix, A., J. Finlay, G. Abowd and R. Beale, *Human-Computer Interaction,* Prentice Hall, 1993.

[5] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems”, *SIGMOD Record,* 18(2), pp.399-407, Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1989.

[6] J. Grudin, “Why CSCW applications fail: problems in the design and evaluation of organisational interfaces”, pp. 85-89 in *CSCW’88 Proceedings of the Conference on Computer-Supported Cooperative Work,* ACM SIGCHI & SIGOIS, 1988.

[7] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice Hall, 1985.

[8] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the Coda file system”, *ACM Transactions on Computer Systems*, 10(1), pp. 3-25, February 1992

[9] J. Mariani and T. Rodden, “The impact of CSCW on database technology”, in *Proceedings of the IFIP Workishop on CSCW*, Berlin, April 1991.

[10] H. W. Thimbleby and David Pullinger, “Observations on practically perfect CSCW”, in *CSCW Issues for Mobile and Remote Workers,* IEE Colloquium, March 1993.

[11] I. H. Witten, H. W. Thimbleby, G. F. Coulouris & S. Greenberg, “A New Approach to Sharing Data in Social Networks”, in *Computer–Supported Cooperative Work and Groupware,* S. Greenberg editor, Academic Press, ISBN 0-12-299220-2, 1991.

---

[1] Branching time interpretations of temporal logic do not use a linear model. However, the branches in the tree represent ‘possible’ worlds and any particular history is still linear (a path through the tree). The major difference between linear and branching time temporal logics is not in the underlying notion of time, but in the interpretation of the temporal operators. In particular, whether the diamond operator ‘eventually’ is taken to mean “there is some possible future state where…” or “in any possible future history there will be a state where…”.

[2] I’ll take the high road and you take the low road, and I’ll be in Scotland before you, ...