Declarative Histories

KEVIN MITCHELL
Agilent Laboratories

Network management systems need to maintain a considerable amount of state in order to react to, correlate and manipulate the properties of the systems under their control. Furthermore, to support root-cause analysis and other diagnostic functions, they frequently need to maintain a historical record of past states as well. In this report we explore some of the issues associated with supporting historical state. We argue that decisions such as which attributes and objects should be stored in the history, with what accuracy, and for how long, should all be expressed declaratively. JDO-style enhancement techniques could then be used to automatically construct the runtime classes, and database schemas, necessary to support this history. Although the paper uses network management as a motivating example, aspects of the proposal may be applicable to many other problems that require archival storage of complex hierarchical state.

Key Words and Phrases: OSS systems, Network Management, Enhancement, JDO

1. IDEALLY...

Network management systems, such as Agilent's NETeXPERT[Agilent 2002], need to maintain a considerable amount of state in order to react to, correlate and manipulate the properties of the systems under their control. This state includes the current topology, with devices like routers, switches, and hubs forming the vertices of the topology graph, and communication links forming the edges. Each of these devices may itself have a considerable amount of state associated with it, as illustrated by the large number of SNMP MIBs supported by some devices. And some of this state, such as the current set of MPLS LSPs, induces further levels of complex hierarchical state. Clearly not all aspects of every device needs to be tracked by an Operations Support System (OSS). The state maintained by any OSS is an abstraction, or an approximation of reality, sufficient to control the system being managed. But as OSS implementations develop, with complex correlation and closed-loop control systems increasingly being deployed, and they become more decentralised, allowing more device state to be monitored efficiently, the amount of data required to describe the OSS's view of the network state will grow considerably.

Maintaining structures representing the current state of a small network within the memory of a single application is a comparatively simple task, if we gloss over the difficulties of collecting the data efficiently. But in practice things are rarely so simple. The state may need to be shared between a suite of applications, or be too large to fit efficiently within memory for a large network. In both cases some form of offline storage, such as a database, will need to be used. Even without space or sharing considerations, some state will need to be stored persistently, for example administrative and accounting details. It may also be desirable to store other aspects of the system's state persistently for performance reasons, to allow a fast restart in the event of systems failure for example. This data will typically be stored in a mixture of databases and file systems. Loading and saving such data, and developing caching strategies to improve performance, can complicate the applications that manipulate the data to a significant degree.

Consider now the situation where a network failure of some form occurs. The exact nature of the failure is not relevant to the current discussion. By the time the failure has been identified, the system state may have changed from that which existed at the time of the failure. Indeed the visible symptoms will often have been caused by an underlying problem that occured even earlier. If we could examine the system state, or at least the OSS's abstraction of this state, at any point in the past then this would help in root-cause analysis. The difficulty in satisfying this requirement, at least in the general case, is one of the reasons that makes root cause analysis so challenging a problem.

For simplicity, we model the (OSS view of) the state of a system by a set of root objects $\mathcal{R}$. We assume that all observable aspects of the state, i.e. the objects and attributes comprising the state, are reachable by following pointers from one or more of the roots in $\mathcal{R}$. The roots might be the objects representing the individual devices in a network, for example. Ideally, given a time t in the past we would like to be able to

Author's address: K. Mitchell, Agilent Labs Scotland, South Queensferry, Scotland EH30 9TG

reconstruct a set of roots $\mathcal{R}_t$ such that they capture the state of all pertinent properties of the system at time t.

We use the term "history" to represent the set of objects representing past state. In this report we explore some of the issues associated with supporting historical state, and suggest a declarative mechanism for maintaining such state. Before exploring the implementation problem in more detail, it may help to first enumerate some desirable properties of a history mechanism. Depending on the application some, or all, of these requirements could be relaxed, although in many cases this may be at the expense of complicating the user application.

- Maintenance of historic state should be transparent. It should be triggered automatically as a side-effect of updating the current state. Similarly the purging of old state that is no longer required should be automated where possible.
- Ideally, the interfaces used to explore historic instances should be identical to those used for the current state. Consider a network topology example. Given a `Router` instance we should be able to access the outgoing links in the same way irrespective of whether the router is part of some past state or the current state. If we define a method to compute all the routers within some number of hops of a specified router then this method should be applicable to all routers, no matter what their "age". The classes used to represent historic state will often need to differ from their "current" counterparts. The mechanisms required to access attribute values are likely to be substantially different, for example. But this requirement for uniformity implies that the the historic and current classes need to be related in some way, either by one being a subclass of the other, both being subclasses of a base class, or by implementing a common interface.
- Where possible, the system should enforce immutability of historic state. We may often want some attributes of an object forming part of the current state of the system to be writable. For example, given an object representing a router interface, we may be able to alter the operational state of the real device by setting the value of an attribute of this object. Behind the scenes, the setter method would trigger a sequence of actions that would result in the device state being altered, e.g. via SNMP. In contrast, the data representing the state of the system at some point in the past should clearly be read-only. Similarly, we might expect the values of some of the attributes in the current state to be able to change under our feet, as they respond to changes in the real device; the corresponding data for a past state should be immutable.
- To allow smooth integration with legacy time-stamped data, we should be able to retrieve the time corresponding to an object in some historic state. To be able to do this for every instance would be prohibitively expensive, and frequently unnecessary, so we need to be able to control which classes support such operations. Note the tension between this requirement and the desire to treat, where possible, historic and current state uniformly.
- We should be able to navigate from an object at time t to the "corresponding" object at an earlier or later time t'. Ideally we should be able to do this without having to manually determine the path to this object from the root(s) of the state, building a complete state for the new time t', and then navigating back down to the new object.
- We should be able to seach backwards and forwards in time looking for changes of interest. And we should be able to express this relatively, e.g. find the time and/or router state at which the set of links for a router r next changed. Or iterate forwards or backwards in time returning all the router instances corresponding to r so we can check some predicate explicitly. Note that such an operation requires careful definition. After all, if we view a change to r as including any change that is visible from r, and the state of the entire network is reachable from r, then we would end up iterating over millions of states before finding one that interested us. So the definition of "all instances of r" would need to be chosen with care in such an operation.

This list is not intended to be exhaustive, or be viewed as essential requirements for all history mechanisms. Some applications might not need some of these features, or there may be ways of achieving similar results by pushing more of the functionality and/or responsibilty on to the user application. But it does illustrate some of the challenges involved in constructing a comprehensive history mechanism.

Suppose we start monitoring a system at time t_0, and we are now at time $t_n > t_0$. Ideally, we would like to be able to access the set of roots $\mathcal{R}_i$ corresponding to any time t_i, where $t_0 \leq t_i < t_n$. Why is this requirement so intractable in general? If one takes the state of the system to include the state of all

the components in the system then there is obviously a considerable amount of state in a large network. Furthermore, some of this state can vary very frequently, e.g. counts in an interface MIB. Periodic snapshots of the entire state may have their uses, but taking a snapshot whenever anything changes is not an option. We can store the individual changes separately, but this creates its own problems. There may be vast numbers of such changes, and to reconstruct the state of the system at some arbitrary point in time may then be quite time-consuming.

In most applications it should be clear that we can only approximate the ideal situation. Accessing past state, with the same precision and detail as live state, for arbitrary times in the past, is just too expensive. In order to make the problem more tractable we need to depart from perfection in various ways. These relaxations may take many forms. For example, we might omit some properties from the history. Some attributes of a router might be accessible in the current state, but not in a prior state. This raises an important issue. There's little point in saving the value of an attribute in the past state if it is never used. Only parts of an application will need to access past state, and their requirements may differ from those components that work exclusively on live data. Of course as the algorithms and heuristics change over time, the amount of state that needs to be preserved historically is also likely to change. It should therefore be easy to change this aspect of the system. We must also be careful not to confuse the need for aggregated statistics with the need to preserve detailed attribute values. For example, suppose we wanted to construct statistics of average link utilization over various time periods. We could keep a detailed history of changes to this attribute, and then construct the statistical results from this history when required. But this would be unnecessarily expensive unless the unaggregated values were also required, e.g as part of a correlation process tracking down the root cause of a network problem.

Another way of departing from the ideal is to reduce the accuracy of attributes in the historical state. For example, consider a parameter recording link utilization. This may fluctuate very frequently, and if every tiny change had to be recorded then the history may become unacceptably large. In many cases it may be sufficient to know only the approximate value for some previous point in time. By "quantising" such parameters when constructing the history, and only recording the changes when the parameter moves into a different "band", we can significantly reduce the state, albeit at the expense of accuracy. The value returned for the attribute in some prior state is no longer necessarily the exact value that the attribute had, but is close to it. Earlier we mentioned that we may not need to maintain a detailed history for an attribute if all we require is aggregated trend data for this attribute. In some applications it may be sufficient to return an average value for an attribute over a time period. By this we mean that retrieving the value of an attribute at time t might return the average value of this attribute over some time period Δt, that contains t. This is a rather different situation than just quantising the value domain, and the cases where this may be appropriate will be rather fewer. In the case of attributes representing monotonically increasing counters then an approach based on interpolation might also make sense.

Having access to an accurate approximation of the system state five minutes or five hours ago may be very useful in root-cause analysis. But much of this value degrades quickly, with there being far less need for such detailed information for the previous day, or week. So another way in which the problem can be made more tractable is to associate "lifetime" information with attributes and classes (or entire states). "*Don't keep anything older than t*" would be a simplistic example of such an annotation. Of course whilst traversing an old state we would presumably want to ensure that it didn't disappear from under us, and so the "garbage collection" process would need to be aware of what was currently being accessed in the clients, or at least a conservative estimate of this. In reality we might want finer control over such behaviour, where some attributes degrade over time, in terms of both their value and timing accuracy, rather than just disappearing completely. For example, for data older than some time difference Δt from the current time we might just store periodic snapshots of part of the state. A request for the value of an attribute in a very old state might then just return the value in the "closest" snapshot. This might be useful behaviour for some attributes, and meaningless for others.

Some temporal ordering may also be unimportant. For example, suppose we make a change C_1 to one object followed by another change C_2 to another object soon afterwards. Recording this fine-grained ordering may be unnecessary. Perhaps just storing the changes simultaneously may be adequate, or even C_2 followed by C_1 if this compacts the history more. In some cases the user application may use transactions to group together such changes, and the storage system should be able to exploit this. But in other cases we may be able to, or need to, manipulate temporal orderings without transactional hints. At a higher level, if we ask the system for the roots at time t then in some scenarios it may be acceptable to return the state at time

t', where t' is earlier or later than t, as long as we have some guarantees about the relationship between t and t'. For example, the time difference is less than some specified bound Δt, or that no event of a specified type can have occured between t and t'.

States may be very large. The key to storing historical states is partly to just store changes whereever possible. For example, consider two states S_1 and S_2, corresponding to times t_1 and t_2. Ideally, if an attribute a of some object has not changed between t_1 and t_2, then we should only have one entry in our data store corresponding to this particular attribute at these times. We say *ideally* because there are many trade-offs here, and in some cases it may be better to duplicate small attributes in order to speed up the process of reconstructing the past state. But, in general, sharing is essential in the storage of history. What is perhaps sometimes overlooked is that sharing may be equally essential when manipulating such states within an application. For example, suppose we retrieve two sets of roots $\mathcal{R}_1$ and $\mathcal{R}_2$, corresponding to the states at time t_1 and t_2. As we traverse these roots, and their descendents, we build objects in memory corresponding to each of these states. In a simple-minded approach the objects reachable from the roots in $\mathcal{R}_1$ may be completely disjoint from those in $\mathcal{R}_2$. However, this may make it difficult to manipulate many states simultaneuously due to the huge memory requirements. Ideally, if an attribute or an object has not changed between t_1 and t_2 then we would like the memory representation to be shared between these two states where possible. We can use "hash-consing" techniques[Appel and Gonçalves 1993] to achieve this in a brute-force fashion, or drive the sharing from more detailed semantic knowledge.

There are a number of things we can conclude from these discussions. Building system that support historical states is clearly challenging and time-consuming. Ideally the classes used to represent historical data and live data should be closely related from a user's perspective, even if, under the hood, they do things very differently. Without such commonality it is hard to write code that uniformly traverses aspects of the live and historical state. The exact representation of the historical state, and the semantics of its objects and attributes, should be heavily influenced by how this state is used. Furthermore, it should be easy to alter such choices as the demands of the user applications vary over time.

In this paper we argue that the behaviour of the attributes in past states should be described *declaratively*. Issues such as whether an attribute should be retrievable in the historical state, and if so with what accuracy, how frequently it is likely change, and similarly for the objects themselves, would be captured in declarative annotations. If the code to support historical state could be driven from these annotations then this would simplify the maintenance of such code, and make it much easier to alter the state definitions as applications evolved. For example, an attribute that was previously not needed in the historical state may now be required, because of a new form of root-cause analysis. Ideally you would just need to change the declarative property of this attribute and "recompile". Before presenting our proposal for a declarative history mechanism, we revisit some of these issues using a concrete example.

2. AN EXAMPLE

We start with a very simple example of a network topology involving routers and links. Using Java, we might define

```
class Network {
  String     name;
  Collection routers;
  ... // Additional attributes
}

class Router {
  String  name;
  Network network;
  Collection incoming; // links
  Collection outgoing; // links
  ... // Additional attributes
}

class Link {
  Router from;
  Router to;
```

```
  int    bandwidth;
  ... // Additional attributes
}
```

Suppose we have a way of monitoring topology changes, for example eavesdropping on OSPF packets, or registering as a client of an IGP Detective server [Lehane 2002]. Updating our representation of the current network state, using the `Network`, `Router` and `Link` classes, in response to topology change events, is a simple task. At any point in time we can also take a snapshot of the current network state using one of a variety of persistence mechanisms. Even for this simple example there are a number of different database schemas that could be used to store this state; Figure 1 illustrates one such schema. Some technologies, such as JDO, could even generate such a scheme automatically.

Network

PK	NETWORK_ID	BIGINT
	NAME	VARCHAR

Router

PK	ROUTER_ID	BIGINT
	NAME	VARCHAR
FK	NETWORK_ID	BIGINT

Link

PK	LINK_ID	BIGINT
	BANDWIDTH	INT
FK	FROM_ID	BIGINT
FK	TO_ID	BIGINT

Network_Routers

PK	NETWORK_ID	BIGINT
PK	NETWORK_ID	BIGINT

Router_Incoming

PK	ROUTER_ID	BIGINT
PK	ROUTER_ID	BIGINT

Router_Outgoing

PK	ROUTER_ID	BIGINT
PK	ROUTER_ID	BIGINT

Fig. 1. Example Topology Support Tables

We now extend this example to include a historical component. There are many ways in which this could be done, and the choice will have an impact on how the application developer can navigate prior state. Whenever the current state changes, e.g. links are added or removed, routers are added or removed, or attributes of either of these are changed, we could record this change persistently, along with the time, real or computed, at which this change occured. But what exactly should we store? Suppose we just saved something akin to the OSPF update messages themselves. This has the virtue of simplicity. Furthermore, some operations on the historical state are very efficiently implemented using this representation. For example, to scroll forwards in time, showing the topology changes in a browser, is very easy. We start with an empty graph, and then apply the changes one-by-one, displaying the graph that results after each change. Having reached some point in time that interests us we can also go backwards in time as long as we have computed, and cached, the "undo" changes at each step. These operations will get slower as time passes, as we have to reapply more and more changes. But we can avoid such problems by taking snapshots of the entire state periodically, and then picking the nearest one of these as our starting point when we wish to reach the state corresponding to a particular point in time. But as we are more likely to want to see recent state, rather than old state, storing the "undo" differences, rather than the updates, is preferable. To reach time t we would then apply the undos to the current graph state, or the nearest snapshot, until we reached t, saving the redo actions as we went along. We could then use these redo actions to visualize the topology changes as time moved forward. A further advantage of storing the undo changes is that it allows us to prune old, unneeded, state in a natural fashion, without the need for snapshots. In the opposite direction, with the changes themselves being stored, we need snapshots to allow pruning.

We have illustrated that some operations on historic state are easy to support, requiring no complex database schemas or run-time support. For some applications, perhaps including the scenarios of most interest to Agilent, such capabilities are sufficient. But what if we want to do something more interesting than just visualizing the topology? For example, some form of complex root-cause analysis involving multiple time instants. The simple-minded approach starts to break down in these cases. One of the reasons for this is that the approach is destructive in nature. To move forwards in time we apply the changes to the graph by updating it. And to move backwards we undo these updates. Suppose we have an object r representing a router at time t, and want to know what it looks like at time $t' > t$ so we can compare the two. If moving forwards in time involves destructively changing r as a side-effect then such tasks become problematic. Old state no longer behaves like an immutable version of the current state. Note that we cannot simply clone r to hold the old value for use in our comparisons as it contains references to other parts of the state which might also be changed. So potentially we would have to clone the entire graph to

preserve the illusion of immutability. This is clearly impractical for a large network, particularly when you want to simultaneously explore the state corresponding to router r, and the state reachable from it, over many different time instances.

The above argument suggests we may need to move away from the "replay" model in such cases, storing changes at a finer level of granularity so we can lazily expand the state for time t as required. For example, if we wish to explore a router instance as it existed at time t we might, at least initially, only construct the router instance itself. If the code then attempts to traverse to the outgoing links then these would also be created dynamically as they existed at time t. And if we then used these links to access the destination routers then these too would also be created. Clearly if the code accessed everything in the state then we would have replaced an expensive clone operation by an even more expensive lazy clone operation. But many forms of root-cause analysis are likely to be much more local, or focused, in their nature. Supporting incremental expansion of the state is clearly difficult, and inefficient, if the state changes are just stored as a linear sequence of update records. But there are many ways in which these updates could be decomposed into finer-grained changes suitable for supporting a lazy creation mechanism.

One approach involves adding temporal information to the tables previously presented in Figure 1. But this is not as simple as it might at first seem. To start with, we need to decide what "kind" of time we want to record. *Valid time* records when a fact was true in the modelled reality, whereas *transaction time* records when a fact was stored in the database. This distinction leads to valid time state tables and transaction time state tables. In some cases we may need to record both kinds of temporal information, leading to bitemporal state tables. For our simple example we are primarily interested in constructing valid time state tables, given our intended use of the historical state. However, if we assume the OSPF updates will be propagated quickly throughout the network than we may be able to gloss over these distinctions, storing transaction time but viewing it as valid time. There are many different ways of adding period information to relational tables. Most databases do not support periods as primitive types, so we typically have to simulate them, either by using start and end times, or a start time and a duration. Further choices arise because these times may represent open or closed periods. As a concrete example, we might model a period by a start and end timestamp, using a closed-open representation; the start time indicating the time at which the row first became valid, and the end time indicating the time immediately after the row was last valid. We model "now" by the maximum time supported by the timestamp datatype for the database being used. So a fact that is currently valid will have a start time some time in the past, and an end time set to this maximum value.

Adding START and END columns to each of the tables in our example seems like a simple step, but there are many complex ramifications of this change. To start with our choice of primary keys needs to be changed. In the case of the `Network` table the `NETWORK_ID` column by itself is no longer a primary key as there may now be multiple rows containing this value, differing in the periods for which they were valid. In this simple example we could just add the START column as an additional key, but the situation is more complex in the general case as time periods could potentially nest. Referential integrity also becomes a more complex issue in this setting. The reader is referred to [Snodgrass 1999] for examples of many of these issues.

Now consider updating a table as a result of an OSPF update event. Instead of the simple UPDATE or INSERT steps that we would use to save the updated current state, we now have to perform an UPDATE to change the end time on the "current" row for this instance, followed by an INSERT to add a new "current" row. Queries over such tables, particularly when joining tables, or searching for times when particular events occured, can quickly reach challenging proportions, especially when they have to be written by hand.

In our simple example we have assumed that the rows representing the current state of the system were stored in the same tables as the rows representing past state. We just need a convention for what to store in the END column to indicate a current row. But in more complex scenarios we might want to separate the current and past state into separate tables, particularly where only a few of an object's attributes, and therefore columns in the current row, are to be preserved in the history. Indeed even for the old entries we might have multiple tables to hold entries of different ages. This would allow us to degrade the information held for an object gradually as it aged. Obviously such techniques would complicate queries even further. Whilst there are obviously performance concerns if taken to extremes, it's likely the bottleneck will be the poor programmer who has to write these queries, at least if generated by hand.

Deleting an object, and therefore a set of rows representing this object in the database, is treated rather differently when supporting historic state. Instead of deleting rows we merely update the END columns to

reflect the current time. These rows will then not form part of the current state. Taken to extremes, the database will just keep growing indefinitely. We therefore also require a scavenger process to enforce the desired lifetimes for each of our objects and attributes. This process would perform such tasks as deleting rows older than T, or setting some columns to NULL for databases where this saves space, or migrating rows to other tables that have fewer columns.

Lazily creating the state for some prior time t helps avoid excessively large memory demands, but there is another equally important side to this problem. Suppose we construct an instance r_1 representing the router r at time t_1 and another instance r_2 representing the "same" router at time t_2. A router may have many attributes associated with it, and some, perhaps most, of these may not have been altered between t_1 and t_2. If we build r_1 and r_2 from the saved state in the database, independently of each other, then there is a danger that the attributes will be duplicated in memory, wasting space. Whilst, for a single router, this may not be too important, it doesn't take many routers, or states, for this to become a large problem. As mentioned earlier, there are a variety of mechanisms we could use to ensure sharing where possible. We might also find that some attributes of a router, such as its name and type, are treated as constants by an application, whilst others change frequently. Factoring out the constant attributes into an auxiliary hidden class in the implementation of the historic `Router` class may increase the potential for space-efficient sharing, but may interact badly with our desire to treat the interface to historic and current router instances uniformly.

What should the runtime instances look like for our example? Suppose they are just like the current instances, except they also have a time embedded in them. Consider a router instance with an embedded time t. Constructing the outgoing links for this object then requires selecting all link instances from the database that existed at this point in time, i.e. where START $\leq t <$ END, and building objects for these links. We also have a choice as to whether to expand the destination router, involving further queries to the database, or leaving it as a placeholder that will get expanded transparently later if required. The link instances may also need to have t embedded in them, particularly if the destination object is to be expanded lazily. This is a simple model, but it stops almost all sharing due to the presence of the time fields. It does, of course, have the advantage of allowing every object to "know" at what time in the history it represents. A more sophisticated analysis of the structure of the state graph may reveal what classes require time to be stored, and those that don't. This could be used to drive the generation of auxiliary classes to promote sharing.

Given a reference to an instance r representing some past state at time t, all the objects reachable from this instance will also be associated with this time. Furthermore, as this state is immutable, at least in principle, we don't have to worry about updates destroying this property. This suggests that we don't really have to attach t to every object reachable from r. We just need to keep track of how we got to an object, and the time associated with the initial root of this path. There are various ways we might achieve this. Consider some code that traverses the state corresponding to this time. Instead of each instance having the time embedded in it, we can pass the time explicitly whenever we wish to traverse from one instance to another, for example when retrieving the outgoing links for a router. Of course explictly having to pass around time in this way breaks one desirable property of a history mechanism, namely that we should be able to traverse past state in the same way as current state. If we are only interested in traversing the state for a single time instant then we can avoid using additional parameters by passing the time implicitly, for example using thread-local storage. But this makes it hard to manipulate multiple time periods simultaneously, a necessity for correlation purposes. We also complicate the caching process. For example, as we lazily expand out the state we do not want to fetch the data twice if we traverse it twice within the same transaction.[1] If we pass time around then the outgoing links cache will now have to be a more complex structure, indexed by time. This raises the question of whether you actually save anything compared to creating a separate instance for each time instant, if you perform such caching.

There will typically be far fewer variables than object instances in a running program, particularly when you are traversing complex hierarchical state. Another approach might therefore be to associate the time with the variables, rather than with the data itself. Obviously for primitive data types this doesn't buy you anything. But for more complex data it might. For example, instead of a variable holding a reference to an edge e it would now hold a reference to an object containing an edge and a time, $\langle e, t \rangle$. This object

[1] Given the historic state is immutable, our caching mechanism doesn't have to respect transactional boundaries. Nevertheless, we presumably need some mechanism to purge state from working memory, and transaction boundaries are just a crude example of a suitable trigger point.

would be an instance of a class satisfying the same interface as the `Edge` "class", and the methods would delegate to the supporting `Edge` class. For example, $\langle e, t\rangle$.getTo() would return $\langle r, t\rangle$, where $r = e.\text{getTo}(t)$. Ideally, instances of such classes would only be bound to variables, never embedded in more complex types. Of course writing such code becomes very tedious, and it is tricky to enforce such constraints. But if the code was generated automatically... With the appropriate static analysis we may even be able to assign the time to a separate variable, avoiding the need to construct the helper instance on the heap. Some of the work on region analysis[Tofte and Birkedal 1998] could perhaps be adapted to this task. The *Visitor* pattern is a common design pattern[Gamma et al. 1995], and visitor generators have been around for a while, e.g. [Bravenboer and Visser 2001; Stirewalt and Dillon 2001]. These usually generate a default DepthFirst visitor with before and after hooks. There is scope for extending such work so that the time is passed around within the traversal code itself. Of course these techniques, whilst reducing the need to embedding time within each instance, still complicate the caching process.

Sharing brings another set of problems. Suppose we have two instances of the same router at different times, and these share an attribute `a`. What if we change the attribute `a` in one of the instances? We would like to treat saved states as immutable objects, but what enforces this, i.e. what prevents errors such as these? Various possibilities spring to mind. We could break the sharing when an update occurs. But this would involve a lot of effort to deal with something that shouldn't occur in the first place. A better strategy would be to make updates on these instances raise an exception, or catch such errors in the type system at compile time. This suggests the need for introducing immutable variants of `Router`, and `Link`.

Many attributes may change their values so frequently that it becomes unrealistic to track every change in an offline database. Suppose the `Link` class had a utilization attribute that was updated either via RSVP-TE messages, or from MIB polling. Clearly its value is likely to change very frequently. In past discussions we have talked about quantizing, or stratifying the values of such attributes. For example, we might just split the utilization into three bands, 0-20%, 21-80%, and 81-100%, and then just record changes whenever the current level crossed between bands. This reduces the rate of change in the history, but at the expense of accuracy. Another way of tackling the problem might be to just store the changes in memory, not persistently. For some attributes the loss of historic state if the machine crashes may be easy to live with, particularly if the information only has a short "use-by" date before it gets discarded anyway. Not all state needs to be stored persistently, and therefore requires an offline database. Perhaps the need for persistent historical state is the exception rather than the norm in terms of the volume of data that needs to be handled like this?

To what extent should time be visible when traversing historical state? Given an object that is part of the state at time t, should we be able to query it to find out what time it corresponds to, i.e. retrieve t? As we have seen, this requirement can be expensive to support, taking up a lot of space, and preventing sharing between states. But when we reach an object with an attribute representing state in an external database, for example legacy data, then we may need to know the time in order to construct a query for this data. So given a network instance n, at time t, can we query at what time this instance corresponds to? Can we do this for any router instance? And any link or attribute of a router? Or are there only some classes which are directly associated with their time, and for the other classes the time must be tracked via the application program. Or by following pointers to other parts of the state that do know the time. So a `Router` instance could find the time by accessing the network and querying that. Similarly a link could find the time by accessing the source, or destination, routers, and from them the network, and so on. Chains may get long, so perhaps some classes may need to cache the time. Perhaps others just don't have access to the time at all, either because they have no need for it, it would be too expensive to store, or that it would be simple to keep track of externally, e.g. in the client's code. And is the code that navigates to the time for a `Router` instance part of the application program, or a method automatically generated in the `Router` class used to represent the historic state? Or is the method in some other class, to allow the time to be queried for all objects, returning "unknown" for most objects?

Given a router instance r representing some time t, should we be able to ask the runtime to find the time when an attribute of r last changed, e.g. a link was added? What if the router disappeared because of a failure. When it comes back is it the "same" router or a fresh one? I.e. when do two instances of a class representing objects at different times denote the same logical instance? And what classes in the model should such a concept be defined on. Clearly it is at best meaningless on many attributes. But for classes representing physical objects such a concept of "sameness" seems essential.

Writing all this code is clearly a very complex task. Furthermore, it can be hard to reconcile all our goals. Particularly our desire to use the same interfaces for exploring current and historic state, which makes it difficult to explicitly pass around time when traversing historic state. However, we only want this commonality to make the programmer's job easier. If we allow the code to be transformed *after* the code has been written, and perhaps additional support classes to be generated automatically, then we might be able to bridge this gap. What kind of technology could be used to implement such a strategy? This is the subject of the next section.

3. JDO

At it's heart we have been describing a process that converts declarative commands about the desired behaviour of historical state into a set of classes that maintains this state. This is reminiscent of what a JDO system does for a single state. For those unfamiliar with JDO we now spend a little time giving an overview of JDO, and then explore how such techniques might be exploited in the context of manipulating historical state.

Java Data Objects (JDO)[Russell et al. 2003] is an architecture for manipulating persistent objects in Java. There are two major objectives of the JDO architecture: first, to provide application programmers a transparent Java-centric view of persistent information, including enterprise data and locally stored data; and second, to enable pluggable implementations of data stores into application servers. Developers can be more productive if they focus on creating Java classes that implement business logic, and use native Java classes to represent data from the data sources. Mapping between the Java classes and the data source, if necessary, can be done by an EIS domain expert. Although the programmer cannot escape from transactional considerations entirely, the code for the persistent classes themselves is not polluted by such issues. JDO defines interfaces and classes to be used by application programmers when using classes whose instances are to be stored in persistent storage (persistence-capable classes), and specifies the contracts between suppliers of persistence-capable classes and the runtime environment (which is part of the JDO Implementation). A JDO enhancer, or byte code enhancer, is a program which modifies the byte codes of Java class files which implement an application component, to enable transparent loading and storing of the fields of the persistent instances. Each access to a persistent field is replaced by a method call that handles the transactional and persistent behavior for the object. The fields themselves are also altered to be private, if not already declared as such, to ensure that any external classes referring to these fields must themselves be enhanced. The input to an enhancer consists of the Java class files to be enhanced together with an XML file describing which classes and fields are to be made persistent, and their mapping to the underlying database schema. There are a number of advantages to using an enhancement-based approach compared to the obvious alternative of a JDO pre-processor. Syntax errors are caught prior to the enhancement phase, and are not obscured by surrounding auto-generated code. A pre-processor also has to deal with erroneous programs whereas a byte-code enhancer can rely on the program having satisfied the compiler. A less obvious advantage of the enhancer approach is that it decouples compilation from schema binding; a compiled program can be targeted at a variety of different database schemas and technologies. A company can sell a compiled class to a customer who can then tailor the class to their particular database environment without requiring access to the source code. Such two-stage deployment is becoming a common and useful technique for selling software components, e.g. the use of deployment descriptors in J2EE-based application servers. One claimed drawback of the enhancement approach is that it can complicate program debugging, as the program being executed does not match the program source in a direct fashion. However, careful design of the enhancement process can minimize this effect quite effectively in practice.

The JDO implementation hides almost all the details of the persistence mechanism from the application. Two primary interfaces are exposed to applications. The main interface for persistent-aware applications is the `PersistenceManager` interface. A persistence manager is responsible for cache management and also provides services such as query management and transaction management. In JDO, objects that are to be made persistent are called persistence-capable objects; they expose the `PersistenceCapable` interface to applications. This provides services such as life cycle state management for persistence-capable classes. The enhancer tool, provided by the JDO vendor, is used to modify the standard output of a Java compiler to add an implementation of the PersistenceCapable interface to those classes that are to be made persistent.

Of course an application cannot be totally unaware of the persistent nature of the data it is manipulating. Access to the data is typically made within the context of a transaction, and the application must also explicitly indicate when an object instance should be made persistent or transient. A typical JDO application

starts by creating a PersistenceManager from a PersistenceManagerFactory.

```
PersistenceManagerFactory pmf = ...
// Initialize factory properties, e.g. database URL and user
PersistenceManager pm = pmf.getPersistenceManager();
Transaction txn = pm.currentTransaction();
```

Continuing our networking example, we could retrieve a persistent instance of the `Router` class, and alter it within the context of a transaction, using code similar to the following:

```
txn.begin();
// perform query
Query query = pm.newQuery(Router.class,"name == \"R1\"");
Collection result = (Collection)query.execute();
Iterator iter = result.iterator();
// iterate over the results
while (iter.hasNext()) {
  Router router = (Router)iter.next();
  // traverse to the first outgoing link and update bandwidth
  Link l = (Link)router.outgoing.iterator().next();
  l.bandwidth = 0;
}
txn.commit();
```

In this example note how the Java code is not polluted by any details of the object-relational mapping. The implicit retrieval of the outgoing link instance is a good example of this. During the enhancement process the enhancer would replace these field accesses by code to retrieve the current values from the database or cache. The compiled class file for each persistence-capable class is augmented with the necessary mapping code by the enhancer application. But how does the enhancer know which fields to persist, or where to store them? A JDO application defines which attributes of a class are persistent using an XML file with the extension `.jdo`. You can provide such a file either for every persistence-capable class or for each package containing such a class. These files define which fields should be persistent and which are transient. In the case of a field representing a one-to-many or many-to-many relation additional information must also be supplied defining the nature of this relationship. Here is an outline of the jdo file for our `Router` class:

```
<?xml version="1.0" encoding="UTF-8"?>
<jdo>
  <package name="...">
    <class name="Router">
      <field name="name"/>
      <field name="network"/>
      <field name="incoming">
        <collection element-type="Link"/>
      </field>
      <field name="outgoing">
        <collection element-type="Link"/>
      </field>
    </class>
  </package>
</jdo>
```

The *.jdo files tell the system what needs to be stored, but not where, or how, to store it. Even for a simple example there are many database schemas that could be used to store the data. We could create one table for each class, with a column for each attribute. This is conceptually simple, but is not efficient when the scope of a query includes subclasses. Or we might create one table for each class hierarchy, rolling each subclass and its attributes into one table. This is efficient for subclass queries, but can potentially result in very wide tables.

Some JDO implementations assume that the database schema is under the control of the implementation. The enhancer not only produces an updated class file but also the SQL code necessary to create the corresponding database tables. When an application has to work with existing data then such an approach is clearly not appropriate. In such cases we need to use a JDO implementation that allows some configurability in the mapping process. One of the ways in which JDO vendors differentiate themselves is in the flexibility of this mapping process.

4. A STRATEGY

At first glance you might thing that JDO is a suitable mechanism for storing historical state. After all, it uses a database... But a value is only stored persistently until the next change. Versioning databases don't really help in this context either. A JDO implementation provides no support for retrieving the roots corresponding to the state at time t. One approach might be to use the JDO framework as a starting point. We would maintain the current state using JDO, and then use the same techniques for building the classes representing the historical state. We would need to develop additional JDO-style description files to declare the required properties of the historical state, and use these to drive the construction of the historical state classes. Initially this would be done "by hand", for an example application, with the experience of this process feeding into a "history enhancer" that could perform the same operations automatically.

To what extent should an application be aware of the difference between historical and live state? For example, historic state is read-only, so given a particular initial class C should this be transformed into two classes, C_h and C_l, where C_h has just getters for attributes, and C_l then extends this with setters? And if so should other parts of the application be able to use/exploit such differences. Or does the historical state look just like the live one, but with setters raising exceptions, for example? And what if C itself contains setter methods. These would obviously need to be replaced in the enhancement process when producing C_h.

Different applications, or parts of an application, may require differing degrees of precision when viewing the historical state. When loading in a state perhaps there should be a way of specifying such things so that the system doesn't waste time and space loading in things that aren't required.

Some systems, such as NETeXPERT, store attributes and classes in a very uniform fashion. This makes it easy to add new classes without altering the database, but is also potentially very inefficient. Whilst such an approach doesn't stop us developing a JDO-style interface to the data, as illustrated in the NESSIE project[Mitchell 2002], the performance penalty of such an approach may be even more of a problem when scaled to historic data. In reality a hybrid approach may be most appropriate, e.g. key structures being mapped to customised tables, whilst other less critical classes are treated in a more generic fashion.

We have also suggested that in some cases the state at time t should be constructed lazily, as a side-effect of the state traversal methods. This is similar to the situation that occurs in a JDO implementation. But taken to extremes this can also be very inefficient. So how much does it make sense to prefetch? For example, in the case of a `Router` instance which attributes should we prefetch? Do we fetch the incoming and outgoing links, initialize the collections with only lazy references to these links, so we can query for indegree and outdegree without further fetches, or not prefetch any link details? Such choices can have a profound impact on the code inside each of the field getter methods, and even on the class hierarchy itself.

To achieve acceptable performance it should be clear that the design of the database schemas and the runtime data structures necessary to maintain the history must complement each other. Furthermore, the many choices and trade-offs in the design space should be driven by an analysis of the annotations in each specific example. This suggests that the enhancement process should produce the appropriate database schemas, in addition to the enhanced classes themselves. JDO implementations also frequently work in this fashion. However, most JDO products also allow the enhanced classes to be tied to existing database schemas, and the expressiveness of this mapping is one of the main distinguishing features of different products. Given the complexities involved in efficient history representation, it would be much harder to map the enhanced "history" classes to existing schemas for maintaining historic state. However, in some simple cases this may be possible. For example, if an existing database already stores simple attribute values, or differences, keyed by time then the enhancement process would just delegate the maintenance of such attributes to the external database. A similar situation would occur where an external application, thread or class provided such information, insulating us entirely from the representation used to maintain this data. To support such a model we may need to define some generic APIs that must be supported by the external code, although most of the example-specific linking would be specified in the annotations used by the enhancer.

Call data records are a good example of such data. Each record stores the history of an individual voice

call. Traversing the state corresponding to a particular instant in time t may then require querying the call data record store. For example, given a gateway you may wish to know which calls were being routed through this device at time t. This would involve identifying which calls were active at this time, which of these were in the appropriate state, and involved the specified gateway. Clearly this is a lot of work, even if done lazily. Depending on the kinds of analysis being performed, and their frequency, it may make sense to store such the records in a form more amenable to incremental loading of attributes and objects. The use of incremental attribute expiry, to reduce the size of the data store, is another example that may require the records to be stored using a more sophisticated structure.

An important aspect of the problem involves the design of a query mechanism for historical data. In many cases the exact choice of schema representation may be hidden from the application programmer. Indeed it may even change over time in response to changes in the demands of the application layer and attribute annotations. So hard-wiring SQL queries directly into the application may be inappropriate. JDO tackles this problem by providing an application-level query mechanism. The queries are expressed in terms of the application's view of the data and the JDO mechanism translates such queries into the appropriate searches on the underlying database tables and run-time caches. The situation is rather more complex in the case of historical state. For example, given a reference to an attribute corresponding to a time t we might want to know at what time $t' < t$ the attribute was set to this value. And if t corresponds to some time in the past history then we might also wish to know at what time $t'' > t$ this attribute was next updated. To answer such questions we need to consult the database, and/or runtime caches of the data. But the precise nature of these queries will depend on how the historical state of this attribute has been represented in this particular application. The query problem is therefore more complicated than in the JDO case, partly because the queries have a temporal aspect to them, and partly because the database schemas we need to work with may be a lot more complex than those required to support simple persistent classes. There is still little standardisation of support for temporal features in mainstream databases, often requiring the schemas and queries to be slightly different for each vendor. This is yet another reason why it is desirable to hide the details of the queries from the application level. To what extent facilities like stored procedures can be used to support these temporal queries is also unclear at present.

The Java community is currently developing a variety of technologies and frameworks to support OSS applications, for example [Sun 2004] and [OSS/J 2004]. Although our proposal is not confined to this area, merely using it as a motivating example, we would obviously need to demonstrate that the approach fitted harmoniously with these other frameworks. There are a variety of spin-offs that might arise as a consequence of the approach described in this document. For example, a new kind of profiling might be useful, examining how the historical state actually gets used, the frequency that objects and their attributes are altered for particular classes, and then using this information to suggest the best annotations to attach to these classes.

A generic tool to help visualize state changes in classes built by the history enhancer might be of great use during the debugging process. It would allow us to extend common debugging operations like "stop when this attribute is next changed" to "tell me the time in the history when this attribute is next changed". Of course as the history is potentially only a loose approximation to reality, the interpretation of such questions has to be treated with some care. Whilst a generic browser/debugger would be no substitute for application-specific, semantics-aware browsers, it might also form a useful starting point for the development of such browsers.

We have been concentrating on history. But many applications need to perform "what-if" analysis as well. What would happen if the network topology changed to ...? Whilst there is usually less need to store such extrapolated state persistently, or to navigate into possible futures, some of the run-time support, treatment of attribute sharing between states and so on, may be useful in this context as well.

We have argued that the traversal of historic state should be similar, if not identical, to the traversal of current state. This places constraints/complications on what we can do, for example time must be largely hidden. We can simplify the implementation of the historic state by allowing the historic state API to be substantially different from the current one, which is really putting some of the complexity back on the user application. Immutability is another example of where we can reapportion effort. Ideally we should either prevent updates of historic state statically, at compile time, or dynamically by raising run-time exceptions. Without such checks we either have to eliminate sharing across time, or just hope that the user application is well-behaved. Enforcing immutability can be tricky to do, so for some applications we might judge the safety net to not be worth the effort.

Whilst supporting a history mechanism, in all its generality, is clearly a complex task, many, perhaps

most, applications do not require such sophistication. So at one extreme an application involving historic state might only wish to access one time instant at a time, only involve simple traversals of the state, sharing no code between new and old state, and move forwards and backwards in time in a linear fashion. For such an application a very simple model of history might suffice. At the other extreme we could imagine a sophisticated correlation/root-cause analysis mechanism that required multiple states to be juggled simultaneously, with traversal not only between elements of a single state, but between corresponding elements in different states, where a substantial amount of code should be able to run on both current and historic state, and so on. In this case the problem is much trickier.

Some of the issues we have discussed suggest likely annotations that would be required by an automated system. The discussion of "sameness" is an example. In other cases, such as the partitioning of classes into those that explicitly contain time information, those where such classes are reachable via fields, and those with no concept of time, there is a clear need for annotational hints, but the exact nature of such hints is rather more vague. And in other cases we do not have a clear enough grasp of the problem to be able to even start contemplating the translation process, and the annotations required to drive it.

This note has made a first stab at exploring the breadth of the problem. Whilst we believe it demonstrates there are potentially significant problems in this area, at least in its most general form, that does not imply that the particular application domains Agilent is working in require such a degree of complexity and sophistication. Others with more relevant domain-specific knowledge will obviously need to judge where in the spectrum Agilent needs to work, and therefore whether this is a problem that would be of interest to the Company. Some aspects of the approach, such as attribute-specific graceful expiry, might be applicable to other applications requiring access to large amounts of data. But it is important to emphasise we are not proposing a new way to store or search very large databases.

In summary, we propose using a JDO-style enhancement technique to

- allow, to the greatest extent possible, a uniform treatment of historic and current state.
- allow past state to be traversed easily, using lazy expansion and caching to support the simultaneous manipulation of multiple past states, for example for root-cause analysis.
- provide flexible mechanisms to abstract attributes, and expire them gracefully, to prevent data volumes becoming unmanageable.

REFERENCES

Agilent. 2002. Integrated OSS assurance - a new-generation OSS solution. Agilent White Paper.

Appel, A. W. and Gonçalves, M. J. R. 1993. Hash-consing garbage collection. Tech. Rep. CS-TR-412-93, Princeton University, Computer Science Department.

Bravenboer, M. and Visser, E. 2001. Guiding visitors: Separating navigation from computation. Tech. rep., Institute of Information and Computing Sciences, Utrecht University.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA.

Lehane, A. 2002. The IGP Detective. Tech. Rep. AGL-2002-9, Agilent Labs.

Mitchell, K. 2002. A JDO interface to OSI Managed Objects. Tech. Rep. AGL-2002-2, Agilent Labs.

OSS/J. 2004. The OSS through Java Initiative. http://www.ossj.org/learning/whitepapers.shtml.

Russell, C. et al. 2003. Java Data Objects. Tech. Rep. 1.0.1, Sun Microsystems Inc.

Snodgrass, R. 1999. *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann.

Stirewalt, K. and Dillon, L. 2001. Generation of visitor components that implement program transformations. In *SIGSOFT Symposium on Software Reusability.* ACM, 86–94.

Sun. 2004. Java management extensions white paper. http://java.sun.com/products/JavaManagement/wp/index.html.

Tofte, M. and Birkedal, L. 1998. A region inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS) 20,* 4 (July), 734–767.