Repository Graph (old)

Feb 2008

David Barrett-Lennard

Site identifiers

It is assumed that each site is uniquely identified with a site identifier. Let S be the set of site identifiers.

Vector times

Let N be the set of natural numbers (ie non-negative integers).

A *vector time* v is a map $v : S \to N$.

Definition: The vector time $v_\varnothing$ satisfies $\forall\ s \in S,\ v_\varnothing(s) = 0$.

Definition: For vector times v_1, v_2, $v = v_1 \uparrow v_2$ denotes the vector time satisfying

$$\forall\ s \in S,\ v(s) = \max(\ v_1(s), v_2(s)\ )$$

Definition: For vector times v_1, v_2, $v = v_1 \downarrow v_2$ denotes the vector time satisfying

$$\forall\ s \in S,\ v(s) = \min(\ v_1(s), v_2(s)\ )$$

Definition: For vector times v_1, v_2, we write $v_1 \le v_2$ if $\forall\ s \in S,\ v_1(s) \le v_2(s)$

Note

- $\forall v,\ v \downarrow v = v$
- $\forall v,\ v \uparrow v = v$
- $\forall v_1, v_2,\ v_1 \downarrow v_2 = v_2 \downarrow v_1$
- $\forall v_1, v_2,\ v_1 \uparrow v_2 = v_2 \uparrow v_1$
- $\forall v_1, v_2, v_3,\ (v_1 \downarrow v_2) \downarrow v_3 = v_1 \downarrow (v_2 \downarrow v_3)$
- $\forall v_1, v_2, v_3,\ (v_1 \uparrow v_2) \uparrow v_3 = v_1 \uparrow (v_2 \uparrow v_3)$
- $\forall v_1, v_2, v_3,\ (v_1 \downarrow v_2) \uparrow v_3 = (v_1 \uparrow v_3) \downarrow (v_2 \uparrow v_3)$
- $\forall v_1, v_2, v_3,\ (v_1 \uparrow v_2) \downarrow v_3 = (v_1 \downarrow v_3) \uparrow (v_2 \downarrow v_3)$
- $\forall v,\ v_\varnothing \le v$
- $\forall v,\ (v_\varnothing \uparrow v) = v$
- $\forall v,\ (v_\varnothing \downarrow v) = v_\varnothing$
- $\forall v_1, v_2,\ (v_1 \downarrow v_2) \le v_1$
- $\forall v_1, v_2,\ v_1 \le (v_1 \uparrow v_2)$
- $\forall v,\ v \le v$
- $v_1 \le v_2\ \&\ v_2 \le v_1\ \Rightarrow\ v_1 = v_2$
- $v_1 \le v_2\ \&\ v_2 \le v_3\ \Rightarrow\ v_1 \le v_3$
- $v_1 \le v\ \&\ v_2 \le v\ \Rightarrow\ (v_1 \uparrow v_2) \le v$

Identification of operations

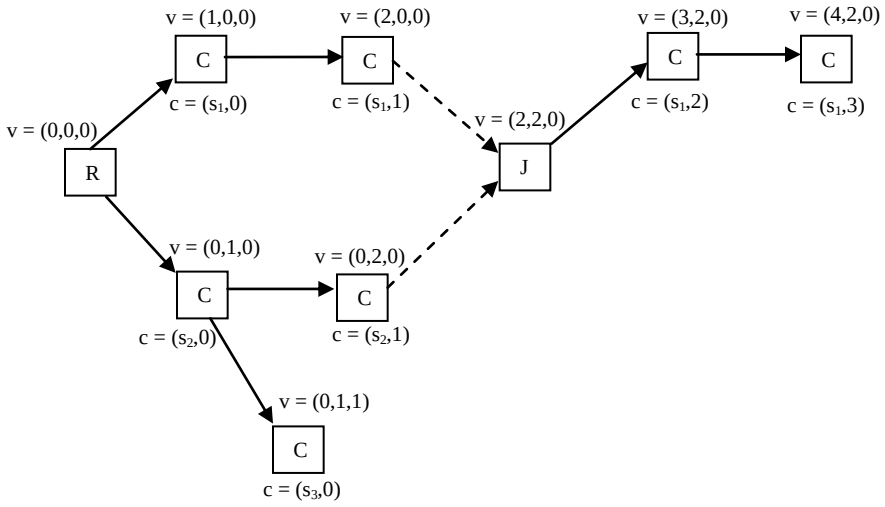
Operations are assumed to be generated at exactly one site. Each site independently assigns a zero based sequence number t to each operation generated at that site.

Definition: Let op(s,t) denote the operation generated at site s with sequence number t.

Introduction

The *Repository Graph* is used to keep track of all operations that have ever been checked into the repository.

The following is an example, where there have been 4 check-ins from site s_1, 2 check-ins from site s_2 and 1 check-in from site s_3.



There are three types of nodes, labeled with R,C or J

R	RootNode	The one and only root node of the graph
C	CheckInNode	Represents the check-in of a single operation by a single site
J	JoinNode	Represents the merge of branches.

There is only one root node (labeled with 'R'). An empty repository is initialised with (only) a root node.

Alongside each node we have provided an associated *vector time*, which in this case is 3 dimensional because there are three sites. The vector time $v = (t_1,t_2,t_3)$ represents the number of operations that have been applied by the respective sites s_1,s_2,s_3 in order to land on that node. The root node has the vector time v = (0,0,0).

Each check-in involves a single operation by a single site. For example the top left check-in was the first operation performed by site s_1, taking the vector time from (0,0,0)

to (1,0,0). The operation itself is uniquely identified using (s,t) = (s_1,0) where t is a local zero based sequence number used by a site s for each locally generated operation.

In our implementation each checkin node records its associated c = (s,t). This together with the graph structure means that nodes do not need to store the associated vector time. Instead algorithms that recurse the graph can calculate the vector time during the recurse.

The example depicts a single join node with a 'J'. It would appear that site s_1 performed an update from a different branch. However the join node is only created when needed for a subsequent check-in of c = (s_1,2). ie a join node is only created when a site is about to perform a check-in, and the site has provided a base vector time that doesn't correspond to any existing node in the graph. There is no validation that the site had previously merged a different branch (because that may have been done with a different repository).

A join node is characterised by its input nodes. ie there is no need to store additional information on a join node.

Overall vector time for the repository
The implementation records a vector time corresponding to all check-ins into the repository. In the above example this is

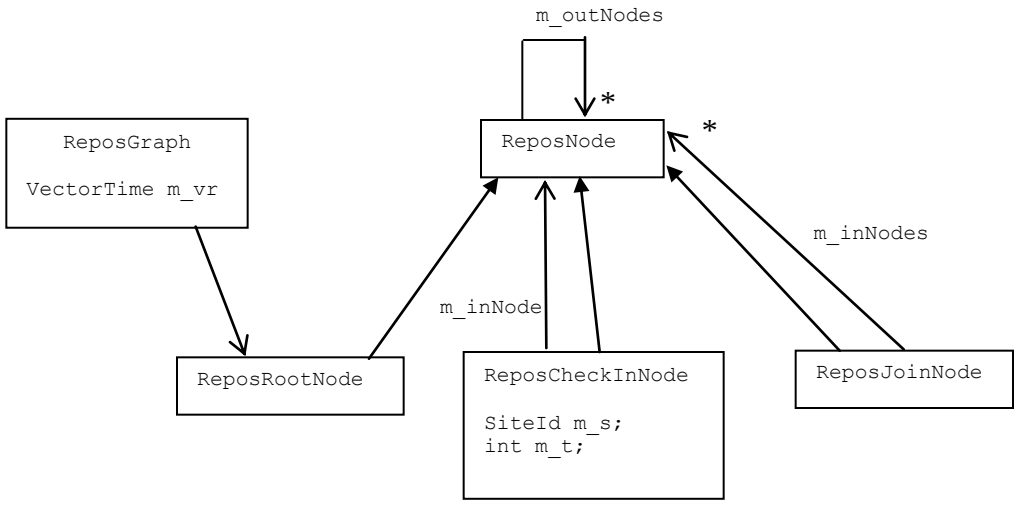
vr(G) = (4,2,1)

There isn't necessarily a node corresponding to this vector time.

Validation
This graph can be used to validate causality in the check-ins. It can also be used to validate the two vector times passed to GetDiff() on the repository. This will help allow the repository to validate all external requests - an important step towards an implementation of a reliable server repository that is even safe against malicious clients.

Scalability
Note that storing a graph of all checkins is reasonable in many practical cases because the number of checkins will not be excessive - a few thousand per year being typical on a project with half a dozen developers.

URL - implementation



Growing the graph for a check-in

A check-in provide the following information

- c = (s,t) of the operation
- The base vector time v

The given base vector time doesn't necessarily correspond to an existing node in the graph. In such a case a join node will be created.

To perform a check-in for given v, a recurse is performed starting from node R. We recurse into a check-in node if and only if its (s,t) is in χ (v). ie if (s,t) is in χ (v) then we recurse into its outnodes, otherwise we ignore the node.

We always recurse through join nodes.

We need to mark which nodes we have already visited, so if we visit a node again we avoid processing it twice. Rather than store a flag in each node the recursive function uses a set of visited nodes.

If none of the out-nodes added an edge to the new node being added to the graph then we should add an edge from the current node to the new node (and in particular this always happens when the current node has no out-nodes - ie it is a leaf node).

Consider that a check-in for the above example is attempted with v=(4,0,0), c=(s_1,4). The repository should not allow this check-in because it breaks causality, because if s_1 has performed 4 operations we know that s_2 must have performed two operations.

To detect this error, when we recurse through a join node we ensure that *all* the input nodes are in X(v) whenever *any* of the output nodes are in χ (v). Otherwise we know that v breaks causality. More formally we have the following definition:

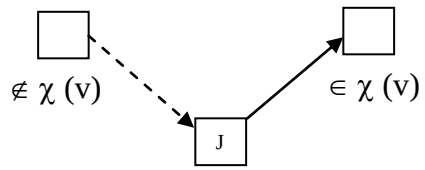
Definition: v breaks causality if

$\exists$ join node n st

$\exists\ n_1 \in$ innodes(n) st $n_1 \notin \chi\ (v)$

and

$\exists\ n_2 \in$ outnodes(n) st $n_2 \in \chi\ (v)$



TODO: Actually we need to formalise more carefully what happens when the innodes or outnodes are themselves join nodes.

The check in of c = (s,t) causes vr to grow by adding (s,t+1).

Validating vector times for a checkin

A check-in provides a vector time v and (s,t) for the operation.

We perform the following validation checks

Error condition	Description	Error generated
$v(s) \neq t$	This ensures that the operation was done in the context of all previous operations generated by that site	Assertion failure
$t < vr(s)$	Given operation is already present in the repository	`REPOS_OPERATION_ALREADY_PRESENT`
$t > vr(s)$	Can't check-in operation because it is not the next expected operation from the site that generated the operation.	`REPOS_OPERATION_MISSING`
$\neg(v \leq vr)$	$\chi\ (v)$ is not a subset of $\chi\ (vr)$. This ensures that the repository already contains the execution context of the operation.	`REPOS_VECTOR_TIME_NOT_A_SUBSET`
v breaks causality	[See formal definition]	`REPOS_VECTOR_TIME_VIOLATES_CAUSALITY`

Brute force validation of checkin vector time

We can check that the number of visited nodes matches the sum over t of the vector time. This proves that there is nothing in the $\chi\ (v)$ that we missed.

However, it doesn't prove that we visited everything we should have. Consider that for a given node we can find all the nodes that precede it. This could be achieved by running a traversal backwards using the in-nodes. Then we could check that all these were visited.

Fast validation of check-in vector time

We avoid the set of visited nodes. Instead we decrement t values in the vector time according to nodes that we have already visited. We can use this to ensure that we visit exactly the nodes in χ (v) - nothing more, nothing less. It is nice to avoid the need for the visited set. At the end the vector time will be empty.

We also need to ensure that v respects causality. There is a problem if there exists a path that starts at a node in χ (v), and ends on a node in χ (v), but somewhere leaves χ (v).

One approach is to recurse the entire graph (ie not using χ (v) to reduce the amount to search). Along a path we have a bool for whether we left χ (v). Once the flag is false we should never come across a node in χ (v). This is kind of a brute force algorithm.

Consider that we store the set of in-nodes. Claim: there is a problem iff there exists a node in χ (v) with an in-node not in χ (v).

Useful testing idea: Consider that every check in node stores the vector time. Then by merging vector times, we have another way to validate v.

Proposal for a more efficient implementation

In practice there will often be long linear branches, perhaps with many hundreds of checkins in a linear sequence.

IDEA: If we introduce linear chain representation then we can avoid a lot of this polymorphism! A linear chain only uses CheckInNodes, and no JoinNodes. Furthermore there is only one out-node along the chain.

IDEA: We strictly alternate between linear chains, and join nodes. A linear chain supports a sequence of check-ins.

```
class CheckInChainNode : public ReposNode
{
public:

private:
    ReposNode* m_inNode;
    std::list<CheckIn> m_checkIns;      // A linear sequence of check-ins
};
```

Note that a ReposCheckInChainNode allows for only one in-node and any number of out-nodes. By contrast a ReposJoinNode allows for any number of in-nodes as well as out-nodes.