Avoiding Undesired Behavior in Concurrent Data Structures

Jeremy Khawaja

May 31, 2017

Abstract

The key to avoiding inefficiency and inconsistency in concurrently accessed structures is having a well-defined concurrent access algorithm. There are many approaches available to use. We devise a a method using spatial and temporal dependency graphs to manage access to arbitrary data structures.

1 Introduction

In this paper we would like to describe an algorithmic approach to avoiding undesirable behavior that arises in systems with multiple processes (threads) accessing a shared data structure. We should first supply a basic definition for a concurrent data structure.

Def. 1.1 (Concurrent Data Structure) - A *concurrent data structure* is a data structure that is concurrently accessed by multiple processes. The processes are allowed to read and write data as well as modify the structure itself.

There aready exist a plethora of approaches to managing access to a shared structure, of which we will make brief mention of only a few techiques e.g. *locks*, *fine-grained locking*, *combining trees*, etc.

For example, here is a basic definition of locking a data structure.

Def. 1.2 (Locking) - A *lock* is a device that enforces a temporary operation ownership of the concurrent data structure by a process e.g. a process in posession of the write lock will be the only process allowed to write data to the structure. Using a set of locks where each lock applies to only a substructure of the overall data structure is called: *fine-grain locking*.

In most programming languages a lock is called a *mutex* (mutual exclusion).

It is also important to specify what exactly we mean by *undesirable behavior*.

Def. 1.3 (Undesired Behavior) - The primary potential behaviors we will be focused on avoiding in this paper are: redundancy, data races, access blocking, and a lack of any consistency guarantee.

For example, some of the potential behavioral problems with locks can vary from things like taking too many locks or too few locks, to taking the wrong locks or taking locks in the wrong order.

Essentially we are trying to avoid problems with inefficiency and inconsistency. Namely, we would like to avoid overwriting values that have already been written. And we would also like to avoid problems that are usually seen as a lack of linearization e.g. a value being overwritten with an out-of-date value.

Of course, every access management implementation will require a degree of tradeoff. For example, are we allowed to read a value from our concurrent data structure at any time or must we always guarantee that all currently-in-progress writes have been completed? We will not focus on tradeoff choices in this paper. How a developer may choose to utilize the algorithm in this paper will depend on the developers personal choice of system requirements. But we do note that the algorithm will be rather flexible in varying the degree of consistency for concurrently accessing a structure.

One of the usual goals of concurrently accessing a single data structure is: computation efficiency i.e. speedup. Sometimes it is also used to avoid having multiple copies of the same structure and being forced to use a consensus algorithm to determine which one has the "true state" of the structure (or a sub-structure) at a given moment in time.

These goals/reasons for concurrent data structure design will not be the focus of this paper. Instead, we will focus on devising an (astoudingly rather simple) algorithm that will be able to 'monitor' and maintain the state of an arbitrary concurrent data structure for us to a degree of consistency that we would like it to have.

2 Concepts

We begin our discussion around the crux of avoiding undesirable behavior: *Ordering* (in the literature: *linearization* and *serialization*).

In our definition of a CDS (1.1), we discussed reading and writing data to a structure as well as changing the structure of the CDS itself. We will label all of these processes as: *access types*. More specifically: these are procedural types. Every time we try to execute an access type we will call this an: *access procedure*.

We do not want to limit ourselves to some predefined set of operations such as "reads, writes, appends, removals, etc.". These do represent a sort of canonical basis for all possible access types, but we

want to be flexible enough to allow for any and all possible methods of accessing a CDS.

Thus, we need to find a way to ensure the linearization, or correct ordering, of access procedures.

For example, when and how will we guarantee that one access procedure completes before another one is started? The how is simple: we can sequentially order our access procedures i.e. apply a total ordering to all access procedures across our system.

But that can be accomplished by using only a single process without the necessity for concurrent processes. Thus we need to address the *when* of ordering access procedures using multiple concurrent processes.

Not all access procedures will need to be totally-ordered i.e. a complete (mathematical) linearization of our access procedures is not always a necessity. It is this *lack of necessity of a total ordering* that allows us to have the possibility of concurrency in the first place.

The degree to which our system can behave non-linearly is exactly the minimal degree to which it can be composed of concurrent processes. (Note: by "system" we are referring to a set of processes plus our CDS).

So, when is ordering a necessity? The only time two access procedures need an order is if one access procedure is *dependent* on the other. We do not specify the reason for the dependency, but if a dependency exists (at all) then it is easy to see that an ordering is necessary.

Since the necessity for an ordering is not always required, we see that a complete linearization of our system is not always a necessity. Rather, the access procedures which carry a dependency on other access procedures create a *partial ordering* for our system.

More specifically: all of our dependencies allows us to have a (disjoint)*dependency graph* for our access procedures.

All good, right?! Sort of. Sadly, this does not resolve all of our problems. We have not discussed exactly how we will order access procedures in terms of processes.

2.1 Spatial Dependencies

First, let's diverge quickly back into a discussion of our CDS. Let's try to formalize it:

Def. 2.1 (Concurrent Data Structure (2)) - A *concurrent data structure* is a graph $\mathcal{G} = \{\mathsf{N}, \mathsf{E}\}$ composed of $|\mathsf{N}|$ nodes and $|\mathsf{E}|$ edges, that is used to index data of type $\mathcal{T}$, and is allowed to be accessed by some set of processes $\mathcal{P}_i$ via some set of access types $\mathcal{A}_t$.

This helps us to remember that a data structure is a spatial object. It carries a certain topology and is invariant through time unless modified. Thus, it is completely possible that while access procedures may not carry a dependency: nodes of our CDS just might! The set of all nodular-dependencies will be called the: *set of spatial dependencies*. These dependencies may be the dependencies that an access procedure "adopts" in order to consider itself dependent on another access procedure.

For example: if access procedure A is trying to write to node $\mathrm{n}_1 \in \mathsf{N}$ and access procedure B is trying to write to node $\mathrm{n}_2 \in \mathsf{N}$, but $\mathrm{n}_2 < \mathrm{n}_1$ (where $<$ is a dependency arrow), then we would want to perform B before A.

This set of spatial dependencies only covers the dependency relationships between certain nodes in our graph. But what if we wanted sub-graphs of our CDS to be dependent or more importantly *independent* of other sub-graphs of our CDS?

Let's define this:

Def. 2.2 (Unique-Independent) - Any sub-graph $\mathcal{S} \subseteq \mathcal{G}$ that does not contain any internal spatial dependencies is called an *unique-independent* of our CDS.

Unique-Independents (UIs) can almost be seen as a *virtualization* of our CDS's spatial topology.

Unique-Independents can obviously have external dependencies i.e. be dependent on each other. Thus, the set of UIs of our CDS forms a dependency graph. More specifically, they form a disjoint directed acyclic graph.

Def. 2.3 (Disjoint Directed Acyclic Graph) - The set of UIs and their dependencies on one another

forms a *disjoint directed-acyclic graph* (ddag) i.e. a disjoint dependencies graph. It is disjointed since UIs are not required to have dependency relations with one another and thus some UIs (nodes) will not have edges in the graph.

Although, since UIs do not have to have other UIs as dependencies it will be simpler to consider: a dependencies list.

Def. 2.4 (Dependency List) - A *dependencies list* for a CDS is the adjacency list of our ddag of unique-independents.

Don't let the name "unique-independent" fool you. There is no requirement for an independent to be spatially-unique. In other words, a UI can share nodes from the CDS with other UIs. The only thing that makes a unique-independent "unique" is that it is in totality unique as to the nodes it applies too.

So while, a UI can be completely subsumed by another UI spatially, it will never refer to the exact same set of nodes that another UI is referring too. And it is this notion of "uniqueness" that allows it to be an independent.

This may seem like a unique-independent that subsumes UIs that have dependencies which are also subsumed UIs, conflicts with our definition (2.2) of an unique-independent.

However, what we have to notice is that a unique-independent is evaluated separately from other UIs i.e. a UI that subsumes other UIs and their dependencies is only allowed to be addressed after all those other UIs have been addressed.

So, while an independent has "no spatial dependencies" it can still subsume UI-dependencies.

One of the most important points to take away from this section is the fact that: the structure of the CDS itself has zero semblance of how we design our concurrent accesses, only the structure of our UI-DDAG (disjoint directed acyclic graph) does.

Now that we have a way to virtualize a CDS using UIs (in a spatial manner), let's tie UIs back in with our previous discussion about ordering access procedures.

We can now use the UI (spatial) virtualization of our CDS to assign an access procedure to a particular UI. We can provide a brief pseudo-code description (written in the Go programming language syntax).

```
type AccessProcedure struct{
        A AccessType
        UID int // unique-independent id
}
```

2.2 Temporal Dependencies

Since our dependency list of UIs based off this virtualization represents all possible spatial dependencies in our CDS it can be used to help us apply an ordering to access procedures. However, a spatial dependency is not the only way access procedures can be ordered i.e. be dependent on each other. The other way access procedures can have an ordering is: *per UI*.

In other words, we can still order all access procedures that have been assigned to the same UI. For example, will read access procedures always take precedence over write access procedures for a particular UI? What if we wanted to be more specific (fine-grained) than that and give *certain* access procedures precedence over *certain* write access procedures?

It would be impossible to suggest that access procedures that apply to a specific UI could always be totally-ordered. Thus, we arrive at the same situation we were in before with trying to linearize our system. Thankfully, we have the answer: a dependency graph (list). Except, this time our dependencies list are access procedures assigned to the same UI.

Whereas we had an UIs dependency list that represented a virtualization of the spatial dependencies, we now have a procedures dependency list that virtualizes the temporal dependencies of our CDS (per UI).

Every UI will have it's own temporal dependencies list and every node in this disjoint dependencies graph will be a specific access procedure.

Def. 2.5 (Temporal DAG) - A temporal directed acyclic graph for a specific UI is the partial ordering of all access procedures acting on the CDS sub-graph at

that UI. Unlike the UI dependency graph, this graph is not (and cannot be) disjointed.

We can also write out any temporal DAG as a temporal dependencies list for a particular UI. In this way we can redefine our access procedure code from above to include a temporal DAG node assignment:

```
type AccessProcedure struct{
        A AccessType
        UID int // unique-independent id
        TID int // temporal dag node id
}
```

2.3 Acidity

We still have to come back around to combining our ordering of dependencies with the processes in our system. But, first we should consider when two access procedures can be simply sequentially ordered. If some set of access procedures which apply to the same UI must be linearly ordered then we can consider the sequential composition of those procedures as being a new access procedure.

This can technically also apply to any sequence of access procedures (assigned to the same UI) that are not dependent on one another. If we want to reduce the total number of individual procedures in our system (i.e. reduce the number of nodes in our temporal dependency graph for a UI) then we can combine access procedures that are linearly dependent (or entirely independent) into a single access procedure.

The allowance for this combination is what we call: *acidity*.

Def. 2.6 (Acidic Procedures) - Any access procedure or sequence of access procedures that appear to the rest of the system as a single operation on a UI are called: *acidic procedures*.

All access procedures *should already be* acidic procedures. But, arbitrary sequences of procedures are not necessarily demanded to be acidic. We want to specifically classify sequences of access procedures that *are* acidic.

Def. 2.7 (Thread) - A *thread* is any sequence of access procedures that can be classified as a single acidic procedure. A thread can also be a sequence of acidic procedures (which may be sequences of access procedures in of themselves).

In terms of a dependency graph: a thread is assigned to a node and is also assigned to the node's dependencies iff the dependent node has only one dependency *and* the dependency has only one dependent. Any sequence of nodes each with at most one dependency and one dependent can be "combined" and considered: a single Thread.

Using threads makes our virtualization complexes more succinct, as any and all sequences of access procedures are either necessarily ordered or are considered single acidic procedures. It also becomes that every node in our disjoint dependency graphs are now a thread.

Prop. 2.1 (Nodes are Threads) Every node in our temporal dependency graphs is a thread (iff we maintain that all access procedures must be acidic procedures).

Cor. 2.1 (Thread DAGs) For every dependency graph of access procedures there is an equivalent dependency graph of threads.

The fact of the matter is that the total number of UIs determines the minimum number of threads required to allow our CDS system to not be missing concurrency improvements (but is usually used when a maximum number of threads is a requirement).

If we choose to use fewer threads than we have UIs, then each thread will likely be assigned to multiple UIs. Can our system still offer a degree of undesired behavior avoidance in this case? The answer is yes, but the management of behavior must be handled internal to each thread.

In this case the threads will need to have internal sequential consistency. In other words, the ordering of dependent operations will only be based on the sequential ordering of access procedures within the thread. So, the thread should be well-ordered according to access procedures that apply to first: UI dependencies then UI dependents.

If the UIs that a thread is assigned to have no

dependency relationship with each other (even composed dependency arrows) then the thread will not need to be ordered internally. Sometimes, ensuring this simple property makes the concurrency of the system simpler to implement and verify.

In the next section of this paper we will discuss how we will ensure acidity for access procedures. But, for now, we simply hold this assumption as a given precondition.

2.4 Dynamic Dependencies

Now that we have methods for defining both spatial and temporal dependencies in our system, the question arises: what happens if a dependency has a lifecycle? What if a dependency needs to only hold true some of the time?

To solve these dynamic situations we can introduce the idea of a *Virtual Dependency Graph* (VDG) which is a dependency graph that can apply to multiple UIs. The key is that it *must* have a limited lifespan. If it does not have a limited lifespan then it just becomes part of a spatial DDAG or a temporal DAG.

Def. 2.8 (Virtual Dependency Graph) - A dependency directed acyclic graph that can apply to one or more UIs for our CDS, but whose lifespan must be less than the lifespan of the CDS (system). Used to define dynamic dependencies in a concurrent system.

A VDG will always have a root node that connects to each UI that the VDG involves. Thus, a VDG can never be a disjoint graph in of itself. The root node represents the final acidic procedure that the VDG will finish before completing its lifecycle.

For example, what if a particular write on one UI needed to happen before a different write on another UI, but that this was *not* necessarily true for *all* writes for these two UIs?

The final write would be the VDG's root node, and the dependency write would be a single child node to the root node. This also means that the two nodes could be a single thread that sequentially ordered a write to the first UI then a write to the second.

So, VDGs can be assigned to a single UI or too multiple UIs.

But, *why* exactly must a VDG always have a limited lifespan (relative to the lifecycle of the system)?

Well, in our previous mentioned example, if that VDG that applied to two UIs lasted the lifespan of the CDS then it would simply be acting as a spatial dependency "representation" (since it would signify that all writes for the dependent UI – that it is attached to – must wait on all writes for the dependency UI that it is also attached to).

And if the VDG had only applied to one of the UIs then it would have represented a permanent temporal DAG for similar reasons. The only thing that makes a VDG dynamic is the fact that it's lifespan is always less then that of the CDS.

Similar to the discussion in the Acidity subsection on assigning the same thread to multiple UIs, assigning a thread to multiple VDGs can possibly lead to an underutilized concurrency potential. Threads that are assigned to VDGs are essentially virtual themselves, as it is their lifecycle that is the lifecycle of a node in a VDG.

Another dynamic property of a CDS system we want to briefly discuss is the idea of: *dynamic UIs*.

If UIs are allowed a lifecycle shorter than the system then the UI dependency graph could potentially need to be constantly redrawn. The only way to outright avoid this scenario is to create another entity that would behave like the nodes in a VDG (and that we specify must have a lifespan less than that of the CDS).

If a UI were to be dynamic, but have dependencies, there would have to be a way to say that it could not end its lifecycle until its dependencies ended their lifecycles (as well as ensuring that it's lifecycle was shorter than it's dependents).

Actually, this leads us to a very important (not yet discussed) lemma:

Lemma 2.1 UIs must cover all nodes in the CDS i.e. every node in the CDS must be addressed by at least one UI at all times.

So, if we allow UIs to be dynamic then we would

have to check that before a UI could end its lifecycle that every CDS node that it addresses must be addressed by some other UI.

Seeing as this would be unnecessarily computationally inefficient, we thus note that dynamic UIs are only feasible if we create the secondary notion of: *Virtual UIs*.

Def. 2.9 (Virtual UI) - A *Virtual UI* is a dynamic UI which has a lifespan shorter than the CDS (UI) or another virtual UI that it has a dependency relationship to.

Because a Virtual UI (VUI) is not dependent on the same premises that real UIs are dependent on, we provide the following lemma to show an important difference between virtual and real UIs.

Lemma 2.2 Virtual UIs do *not* need to be unique (either relative to real UIs or other virtual UIs) in the sense that real UIs are "unique" relative to other real UIs.

This lack of a necessity for any uniqueness of VUIs is what gives them a significant degree of flexibility in use (even if there use can only be relatively temporary). Also, unlike VDGs which mimic Temporal DAGs, VUIs (and their dependency relationships) are more of a dynamic version of spatial UI DDAGs.

This allows us to have: *Virtual UI Dependency Graphs*. But, then this also implies that every VUI could potentially have a VDG attached to it (that must have a lifecycle less than or equal to the VUI node itself). All VUIs must have a lifecycle less than or equal to the VUI they are dependencies of. VUIs can apply to any set of nodes, but they must either have at least one real or one virtual dependent.

Every VDG and VUI node is a thread that must make it's dependents aware of its existence. In fact, all nodes in any dependency graph (virtual or real) are just threads. The only difference is that real thread nodes do not control declaring themselves dependencies of one another (this should be controlled by a "main" process), only virtual thread nodes can declare themselves as a (temporary) dependency of other thread nodes.

The final thing we would like to mention in this dependencies section is the fact that virtual (dynamic) nodes (both VDG and VUI nodes) can still have real nodes (spatial and temporal) as their dependencies. This implies that VDG nodes and VUI nodes with real dependencies need to be checked for cyclic dependencies, as it essentially becomes a (temporary) extension of the permanent spatial and temporal dependency graphs for the CDS. In other words, all extensions of any of our graphs must always preserve them as being DAGs (even if disjoint).

2.5 Invariance

Another part of discussing disjoint directed acyclic graphs and their equivalent dependencies lists is determining if an access type is allowed or not allowed to be used on a specific UI (or even at a specific time).

Boundary Nodes on our UI dependency graph can be useful for determining what access types are allowed for a UI. For example, it might be simpler to allow CDS-structure modifying access types for boundary UI nodes only: specifically appending and pruning sub-graphs to the CDS.

But, we should first formalize what boundary nodes are and then look at an example of how this can be useful.

Def. 2.10 (Leaf Boundaries) - Any real DDAG node which has no dependencies is considered a *leaf boundary* node on that DDAG.

Every real (spatial and temporal) node can be represented by 4 numbers: spatial dependents, temporal dependents, spatial dependencies, and temporal dependencies.

```
type RealNode struct{
        SpatialDependents uint
        TemporalDependents uint
        SpatialDependencies uint
        TemporalDependencies uint
}
```

The basic difference between leaf-boundary and non-boundary real nodes can be expressed with these four variables succinctly:

- Spatial Nodes: {w, x, y, 0}
- Spatial Leaf Boundary Nodes: {0, x, y, 0}
- Temporal Nodes: {0, x, y, z}
- Temporal Leaf Boundary Nodes: {0, 0, y, z}

There is another kind of boundary node for real DDAGs:

Def. 2.11 (Root Boundaries) - Any real dependency graph node which may or may not have dependencies but does not have any dependents is considered a *root boundary* node on that dependency graph.

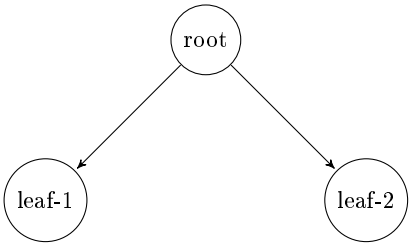
The classes for root boundaries can be expressed using our primary 4 variables succinctly as well:

- Spatial Root Boundary Nodes: {0, 0, y, z}

Notice, that we do not have Temporal Root Boundary Nodes defined. This is because of the following corollary.

Cor. 2.2 (No Temporal Root Boundaries) There are no Temporal root boundaries as every Temporal node must have at least one Spatial node dependency.

Example 2.1 (2-Regular Tree): We will consider a 3 node 2-Regular Tree for which we assign one node to a UI and make the root node's UI dependent on the the two leaf nodes UIs. This makes our UI DDAG have one root-boundary and two leaf-boundary nodes.



In this case, we will allow structure appending access types on the two leaf nodes. Specifically, we would like to ensure that these two UI nodes can only create new nodes on our 2-regular tree for each of the two branches of the tree (so that each UI has control over its own half of the 2-Regular Tree).

Thus, we can help ensure that the tree remains 2-Regular by not giving the root boundary UI node any append sub-graph access type rights.

In Example 2.1, we see that modifying the structure of our tree is dependent on what we want to preserve (which in this case is a 2-Regular Tree structure). Thus, we did not allow the root boundary UI with it's dependencies to have the ability to use access types that modify our CDS structure (as there was no need for it too).

We divided the tree into 3 UIs which established an easy way to divide access procedures as well. However, we still would not want any of our 3 original CDS nodes to be deleted, *yet* simultaneously we do want to be able to remove CDS nodes that have been appended to the two leaf CDS nodes if we want too. Therefore we still need a way to more precisely determine access type allowances.

Afterall, the above example is a very simple example. What would happen if we wanted to apply access rules for an arbitrary CDS graph structure or a more complex UI DDAG? As we have seen: one way to define a rule is by defining what is we want to preserve (at a given time) and what is allowed to change.

For example, we did not want the original three CDS nodes in Example 2.1 to be deleted. But, we could have if we wanted too.

In fact, each UI could have been assigned a list of *immutable* CDS sub-graphs that cannot be removed or even have their values modified. More specifically, we could have had a list of sub-graphs in the CDS that are *invariant* to certain access types.

Def. 2.12 (Invariants) - Any data d stored in our CDS, or any subgraph $\mathsf{S} \subset \mathsf{G}$ that does not change when acted upon by a specific Access Type $\mathcal{A}_t$ is called an invariant of that access type.

$$\mathcal{A}_t(\mathsf{d}) \rightarrow \mathsf{d}$$

$$\mathcal{A}_t(\mathsf{S}) \rightarrow \mathsf{S}$$

Note 1 – Immutability is only a kind of invariance. A low-resolution invariance which blanket-denies all modification access types.

Every node in our CDS Graph G has a set of access types $\mathcal{A}$ that it is invariant under. Either the CDS node and its value remain invariant under an access type (e.g. a “read” or "copy" access type), or only the node itself remains invariant (e.g. a “write” or "increment" access type). Otherwise, the node and

the value are mutable (e.g. in a "delete-subgraph" access type).

Thus, these are the canonical invariance classes for access types:

- Node and value are invariant.
- Only node is invariant.
- No Invariance.

We could of course describe invariants at a higher level e.g. in terms of sub-graphs which are or not invariant under an access type. But, for now we just leave this as a pattern that can be used to help determine access types that are allowed (or not allowed) to be used by a particular process accessing the CDS.

2.6 Partial Orderings

The last thing we want to mention about dependency graphs is rules for how the graph structure should be organized. One approach is by defining a partial ordering for the nodes that will be in the graph. We will not expound completely on this topic in this paper, instead only mention a single possible approach to temporal DAG orderings.

We want to discuss *access type priorities* and how they can be used to define an ordering on a temporal DAG. We know that access types can be simple reads and writes to much more complex calculation-based overwrites. However, our example will focus on simple read and write priority operations in regards to implementing a degree of "consistency".

By consistency we are referring to determining how often a read can be guaranteed to be reading after the latest write. One obvious simple way is to specify that if any write access procedures are in progress that a read must wait for them to signal completion before reading a value. This of course translates to a simple dependency graph where reads become dependent on write access procedures to complete before commencing themselves.

The way in which we can implement this is via: access type priorities. If the write access type has a higher priority than a read, then every write is added as a dependency to any existing read operation for a particular UI. Priorities can also vary from UI to UI.

The key point is that priorities of access types automatically grants us an ordering for access procedures for a UI (or for an entire CDS if applied to all UIs). And we get an automated ordering if all access types receive an ordering.

2.7 Review

Def. 2.13 (Spatial Threads) - *Spatial Threads* are threads that last the lifespan of the CDS and are attached to a single UI.

Def. 2.14 (Temporal Threads) - *Temporal Threads* are threads that last the lifespan of the CDS and are attached to a single UI (in this case temporal threads represent all (permanent) threads beyond the initial spatial thread attached to a UI).

Def. 2.15 (Virtual Threads) - *Virtual Threads* are threads that have a lifespan shorter than the CDS (or "Thread Node" that they are a dependency of) and can be assigned to multiple UIs.

3 Code

How are we going to assign unique-independent modifications to a thread? In other words, how will we ensure that a thread is only allowed to make modifications to a single unique-independent?

We will also address the atomicity of operations and sequence of operations as defined *internally* to a thread.

Since a leaf boundary node is a node without dependencies it can always iterate through its atomic operations infinitely often without waiting for other nodes.

3.1 VDGs

Def. 3.1 (Virtual Dependency Graph) - A temporary secondary dependency graph, which behaves independent of our main dependency graph, and only behaves on some given subset of UIs.

A VDG is basically equivalent to an internal partial-

ordering of operations within a thread.

Except, that if we consider a VDG a thread, then we have to consider it as a thread node which has no dependents or dependencies. Thus, a VDG is like a virtualized thread node that allows us to manipulate multiple (V)UIs instead of only a single V(UI).

VDGs can present a potential concurrency-safety issue if you use them without taking into consideration the assumptions of what they are.

Make sure that all VDG operations are safe to operate independently from the main dependency graph.

In summation, using a VDG is making the following assumptions – all access types that the VDG will be using are okay to be used without dependency management from our graph. As well as the fact that all VDG operations may occur at any given time independent of our global graph. They are simply a temporary secondary dependency graph that will operate on our CDS independent of our main graph.

3.2 CDS Mutation

The answer to CDS mutation is that the ordering of nodes as chosen by the developer should always be such that mutation access procedures are ordered before or after a read-value or write-value operation e.g. the priority of a read-value and write-value procedure could be greater than the priority of a mutation procedure.

So, while it is possible to use fabric to perform dirty reads and writes, etc. It is not the purpose of fabric to be full-proof, but rather to empower the developer to more easily implement full-proof (as defined by the spec of the system) concurrency control in their software.

Actually, the purpose of behavior avoidance is *not* to disallow a behavior from being designed into the system, but rather to be able to verify more easily that a certain behavior (or lack of behavior) exists in the system design.

4 Memory Management

Can we utilize this new model to find a better method for Memory Management of Concurrent Data Structures (in runtimes that are not using a built-in GC)?

Garbage Collecting Virtual DDAGs becomes necessary.

5 Formal Verification

Formal verification is entirely dependent on the system requirements and design. This is not (per say) an algorithm that can be glued into existing projects, but much more so a library that follows fundamental mathematical principles that can be utilized in designing a CDS system (a system with CDSs within itself).

We could still use a formal verification section in the paper that briefly discusses using the ideas found in the paper (e.g. in the Fabric package) to verify that a certain CDS system (CDS + Accessing Processes) either maintains a certain defined behavior or avoids a certain defined behavior.

The section will describe a rudimentary approach to: defining approaches to verification.

For example, if we have a CDS system that we design using the ‘fabric‘ package (or a similar dependency) ...

5.1 Built-In Verification Methods

There are already some built-in verification methods in the fabric package. For dependency graphs we check things like:

- CycleDetect(): which ensures that the graph is a DAG
- TotalityUnique(): which ensures that all UIs are totality-unique
- Covered(): which ensures that every node and edge in the CDS has been addressed
- CheckVUIDependents()

6 Conclusion

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.