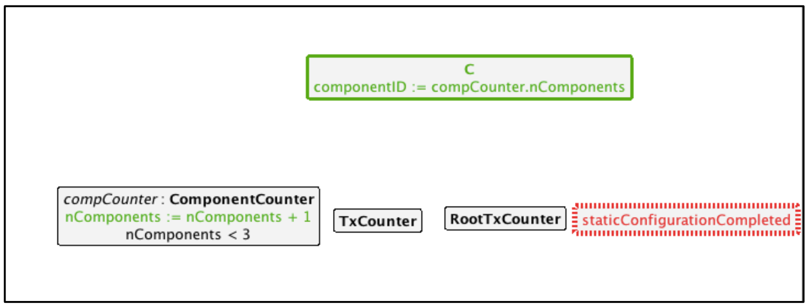
Graph Grammar for Version Consistency

This document collects the rules showing the evolution of dynamic dependencies on the basis of the distributed management algorithm.

1. Definition of the Static Configuration

This set of rules support the definition of a static configuration with an arbitrary number of components and interconnections. Once the definition is completed rules StaticConfigurationCompleted can be applied to proceed with the execution of the algorithm

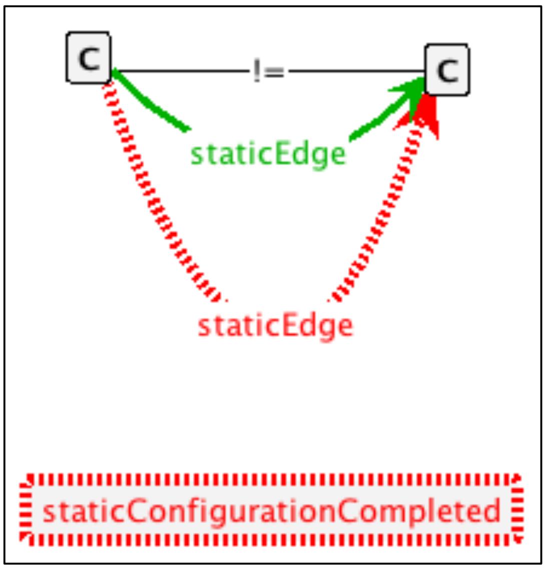
Rule 1: AddComponent


C
componentID := compCounter.nComponents
compCounter : ComponentCounter
nComponents := nComponents + 1
nComponents < 3
TxCounter
RootTxCounter
staticConfigurationCompleted

Description:

If the static configuration is not completed, it is possible to add new components. Each component has an unique id starting from 0. The node compCounter keeps track of the number of created components to maintain the id of the component unique (the first created component has id 0, the second id 1, …). TxCounter is used to count the total number of transactions.

Rule 2: AddComponent



Description:
If the static configuration is not completed, it is possible to add a static edge between two components (two distinct instances of type C).

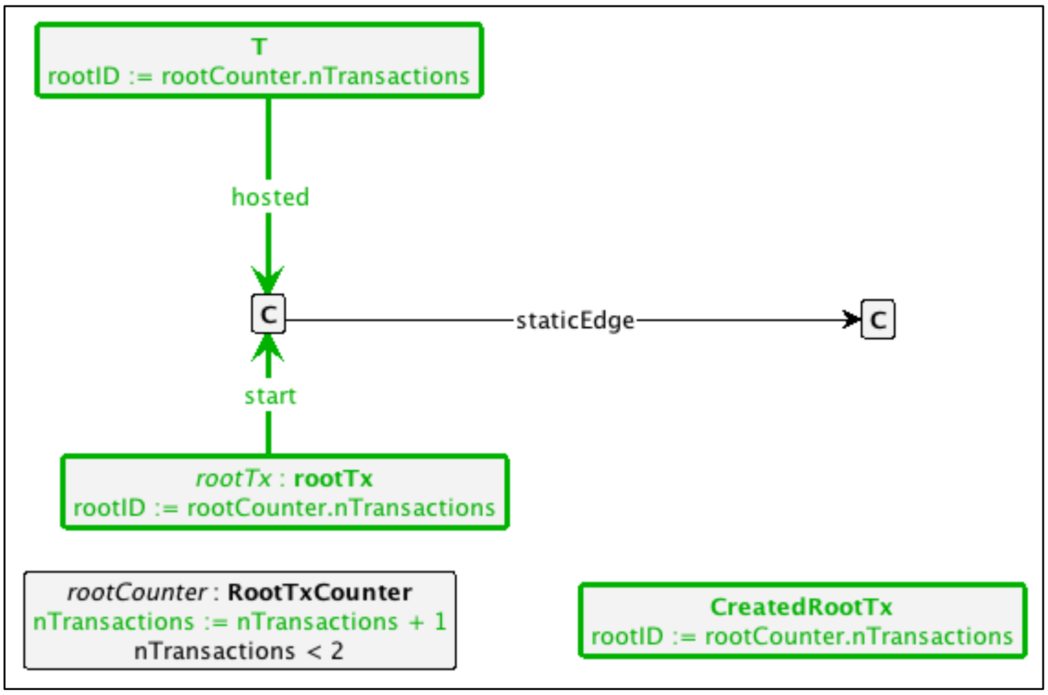
Rule 3: StaticConfigurationCompleted

staticConfigurationCompleted

When the definition of the static configuration is completed we can fire this simple rule. It represents a flag that avoids that new components are added or connected while the algorithm is running.

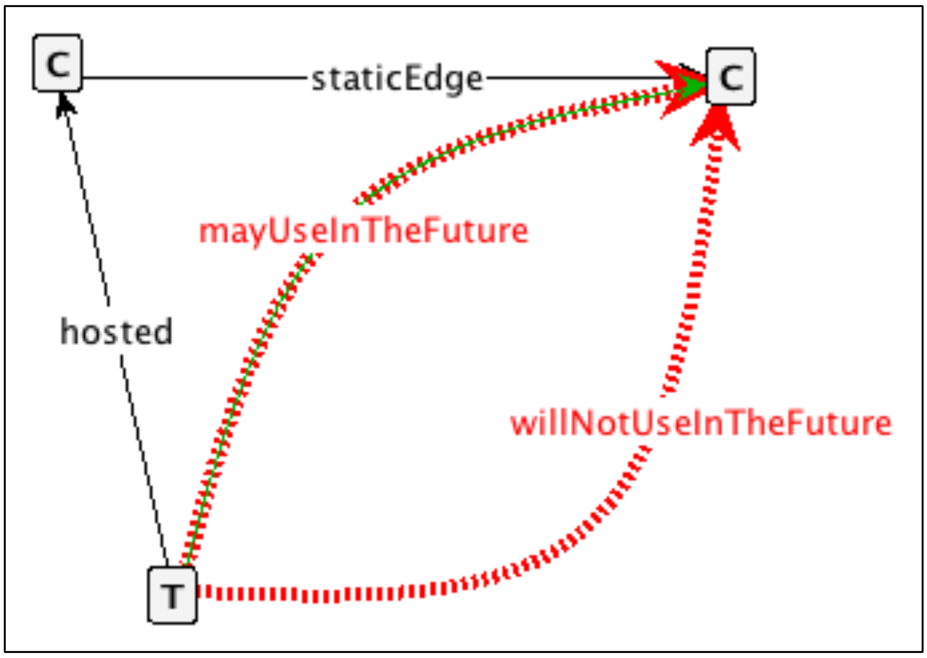
2. Modeling transaction

Rule 3: StartRootTransaction


T
rootID := rootCounter.nTransactions
hosted
C
staticEdge
C
start
rootTx : rootTx
rootID := rootCounter.nTransactions
rootCounter : RootTxCounter
nTransactions := nTransactions + 1
nTransactions < 2
CreatedRootTx
rootID := rootCounter.nTransactions

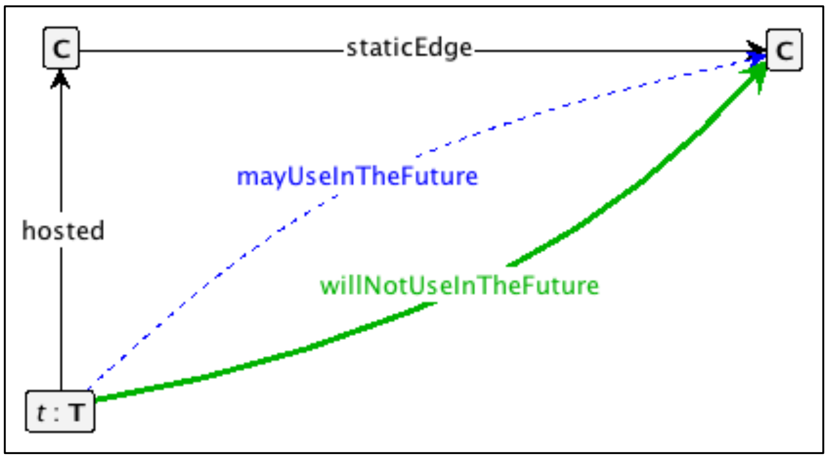
Description:
A root transaction rootTx can start in any component. It is identified by rootID that is unique thanks to the global counter rootCounter. The consequence of the initiation of the root transaction is that the component hosts a Transaction with rootID equals to the id of the node rootTx. In this way we can also specify that a rootTransaction is also a transaction. To reduce the statespace, we limit the initiation of the root transactions to component that has at least one outgoing static edge.

Rule 4: addToFutureSet



Description:
A transaction represents a local part of the logic of the application.
The information we need for the algorithm (and that we can infer by the static analysis of the code) regards the possibility of invoking a certain component in the future. This is modeled with an arc called **mayUseInTheFuture** starting from a generic transaction to a statically dependent component.
The opposite condition, that is that a transaction will never invoke in the future a certain component, is modeled with the arch **willNotUseInTheFuture**
By adding the arcs mayUseInTheFuture and willNotUseInTheFuture, we model the insertion and removal of components in the Future Set.
This rule states the following: if **mayUseInTheFuture** is not already present and if **willNotUseInTheFuture** is not present, an arc mayUseInTheFuture can be added to the graph connecting the transaction to the component which may be used in the future.

Rule 5: removeFromFutureSet

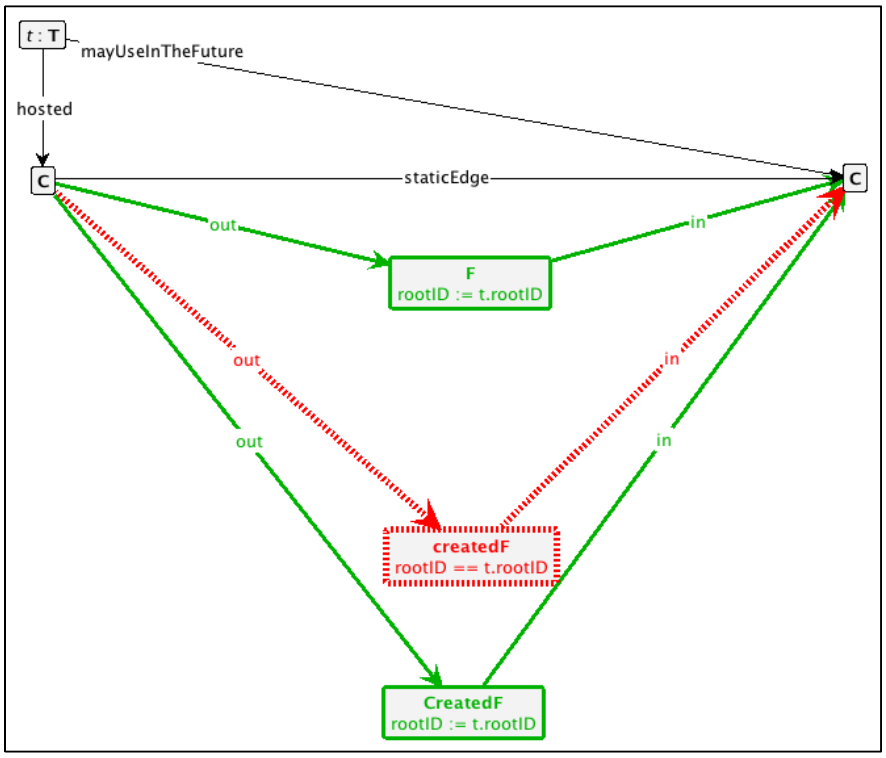


Description:

This rule is the dual of Rule4 and models the removal of a component from the future set. It states the following: If a component will not be used anymore by a given transaction, the arc **mayUseInTheFuture** is removed and the arc **willNotUseInTheFuture** is added.

3. SetUp step

Rule 6: addDirectF



Description:

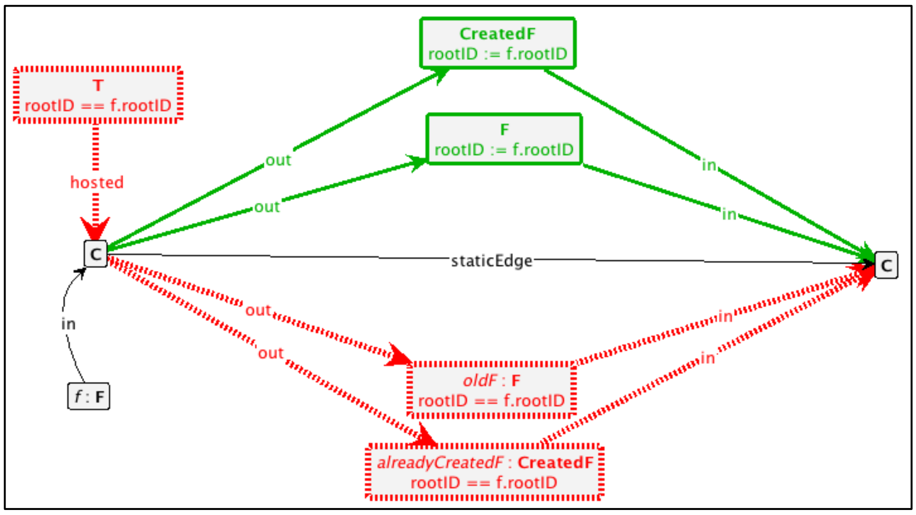
This rule is applied for component which just initiated hosting the root transaction. The root transaction just started and therefore it is possible to know which components can be used in the future.

A future edge can be added from one component hosting a transaction to another component if:

1) it does not already exist a future edge linking the same components and labeled with the same rootID.
2) The component sending the message for adding the edge is hosting a transaction and that transaction may use in the future the component receiving the message.

Only for the purpose of the proof, we also remember the event of the creation of the future edge with the arc **createdF.**

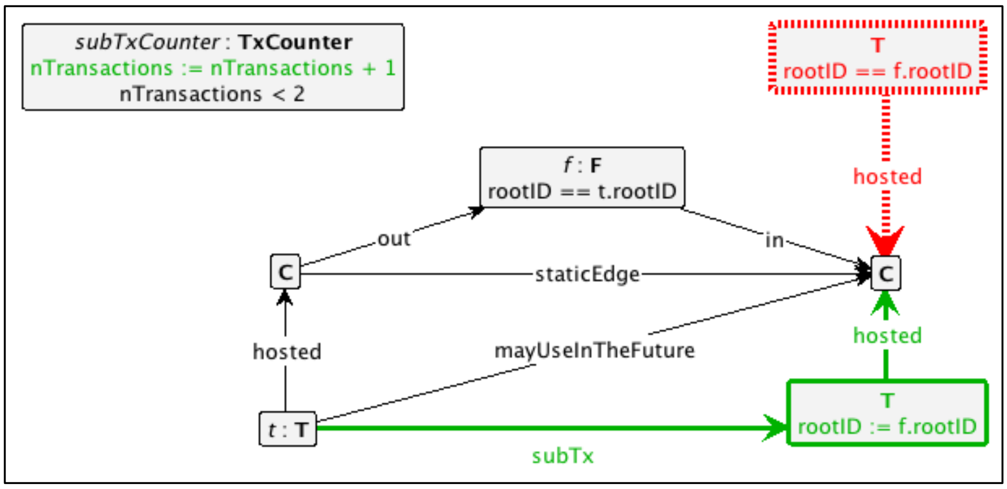
Rule 7: addRecursiveF



Description:
This rule is applied for “recursively” propagating the future edges.
Whenever a component received a future edge, it immediately propagate the edges to all the other statically dependent components.
The combination of Rule 6 and Rule 7 defines the setup step. As in rule 6, also in this case we remember the creation of future edge with createdF.

4. Progress step

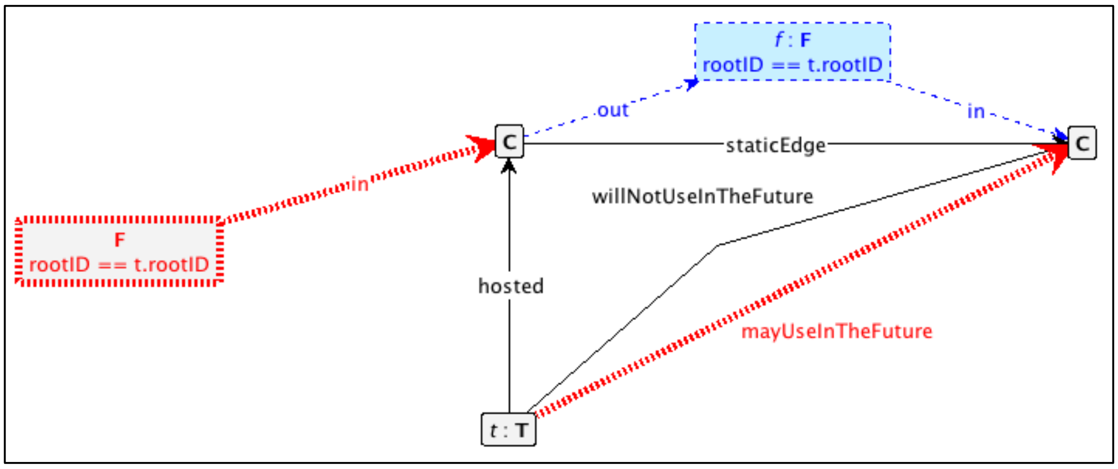
Rule 8: startSubTx



Description
A new subtransaction of a given transaction t can be initiated if:

1) there exists a static edge between the two components
2) there exists a future edge between the two components
3) (to limit the statespace) **no** transactions with the same root id is currectly hosted by the target component.

Rule 9: RemoveF

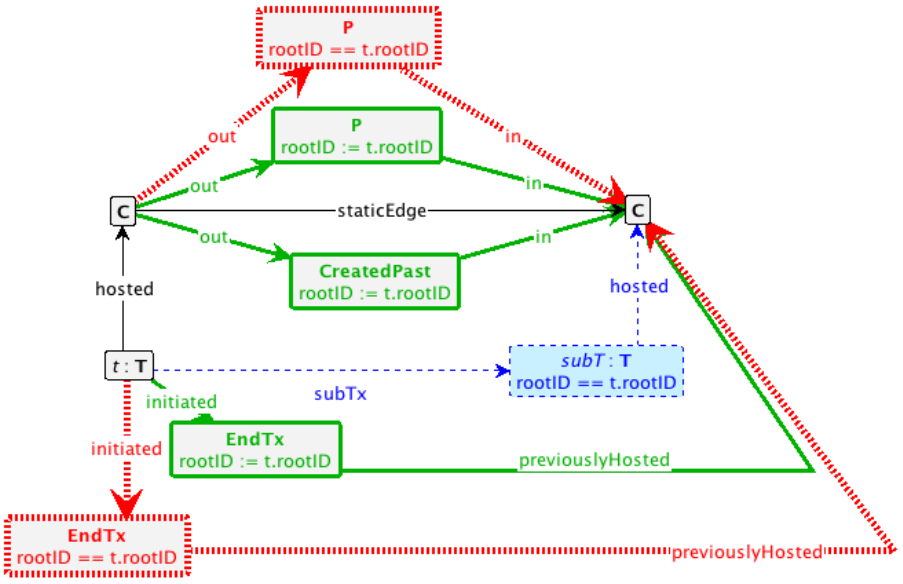


Description:

A future edge between two components can be removed if

1) the source component is hosting a transaction that willNotUseInTheFuture the target component (see Rule 5)
2) The source component does not have incoming future edges labelled with the same rootID

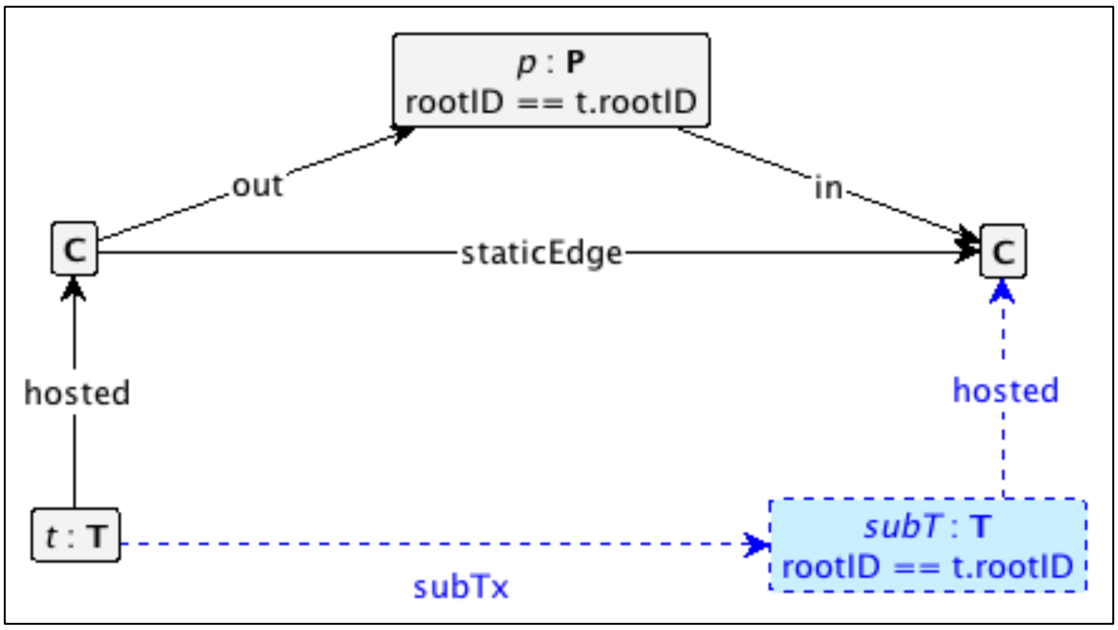
Rule 10: endSubTx_addPastEdge



Description

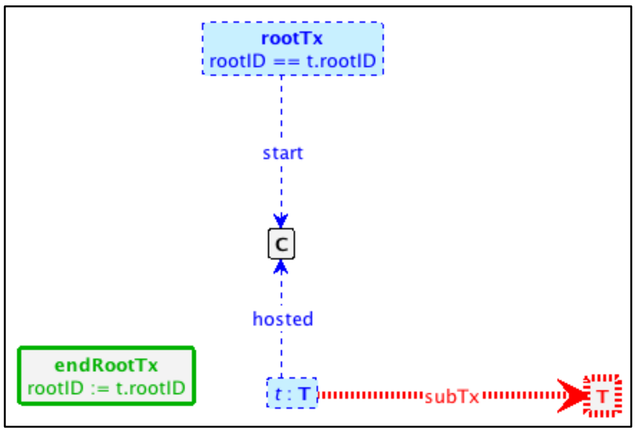
If a subtransaction terminates, a past edge is created from the initiating component to the target component.
For model checking we also remember the event of the creation of past edge with CreatedPast.
Moreover, we remember the first time a subtransaction ends with the node EndTx.

Rule 11: endSubTx



Description
If the past edge is already present only the arcs and nodes related to the subtransaction is removed and no other past edges are created.

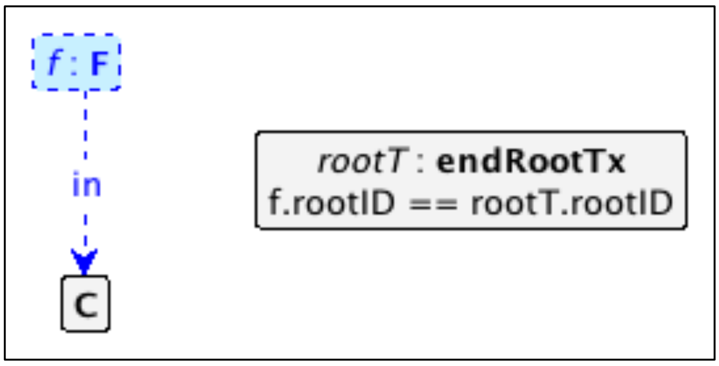
Rule 12: endRootTx



Description:
A root transaction can end (and thus removed from the graph) only if there are no other initiated subtransactions in execution. We remember the end of the rootTx with the node endRootTx.

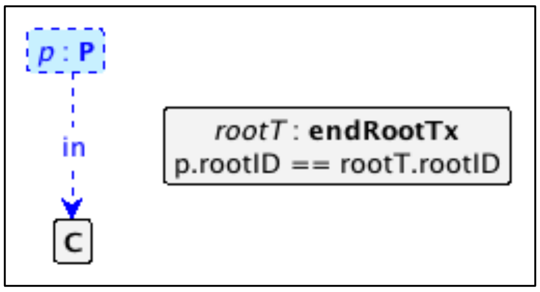
5. Cleanup step

Rules13, : removeDanglingF



Description:
When the rootTx terminates (endRootTransaction) we remove all the future edges with the rootTx.rootID

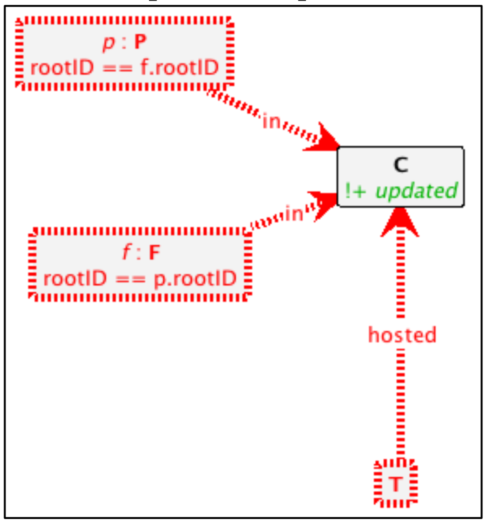
Rule 15: RemoveDanglingP



Description:
As for future edges (rule 13) remove dangling past edges if no rootTransactions are in execution(rule 15) or if the rootTransactions in execution have a different rootID w.r.t. the past edges to be removed (rule 16).

6. Freeness condition

Rule 17: updateComponent



Description:
A component is free, if it is not hosting any transaction (this substitutes the original condition on the self edges), **and** there are no incoming past and future edges labelled with the same rootID.

Discussion on how the grammar satisfies the properties of dynamic dependencies.

1. (host-validity) Given a transaction T, the hosting component C must have a dedicated pair of edges $C - \frac{future}{root(T)} - > C$ and $C - \frac{past}{root(T)} - > C$ during the whole duration of T (self-edges).

We remove the notion of self-edges and change the freeness property accordingly. Therefore we do not need anymore this property

2. (locality) Each future or past edge $C - \frac{future(past)}{T} - > C'$ must be paired

with a static edge $C - \frac{static}{} - > C'$;

Proof. The property holds by construction. The creation of future edges are performed as a result of the application of Rule 6 and Rule 7. In these rules a future edge can only be created if it exists a paired static edge. The same is true for past edges w.r.t. to Rule 10.

3. (future-validity) A future edge $C - \frac{future}{} - > C'$ begins to exist no later than the first time a transaction T′ ∈ ext(T) is initiated, and continues to exist at least until no transactions hosted by C will initiate further T'' ∈ext(T) on C' through $C - \frac{static}{} - > C'$;

Proof. Let us divide the proof in two parts. In the first part we prove that a future edge $C - \frac{future(past)}{T} - > C'$ begins to exist no later than the first time a transaction T′ ∈ ext(T) is initiated. This is true because the creation of future edges occur only in the setup step (rule 6 and rule 7). In this step no subtransactions have been already initiated on the target component. Moreover, the initiation of the subtransaction occurs in rule 8 which requires already the presence of a future edge (labeled with rootID of the subtransaction to be initiated) between the source component and the target component. In the second part we prove that a future edge $C - \frac{future(past)}{T} - > C'$ continues to exist at least until no transactions hosted by C will initiate further T'' ∈ext(T) on C' through $C - \frac{static}{} - > C'$. The removal of future edges occurs in three rules: Rule 13, Rule 14, and Rule 9. Rules 13 and 14 are applied during the cleanup step. This means that the root transaction T has already terminated. According to Rule 12, no other subtransactions T'' ∈ext(T) can be in execution on C' and, given the fact that T terminated, according to Rule 11, no further subtransactions can be inititated. A future edge can be removed also during the progress step in Rule 9. However, Rule 9 to be applied (and therefore to remove the future edge), requires that no future edges are incoming on C. If no future edges are incoming to C, according to Rule 8, no further transactions can be initiated on C. After the application of Rule 9, again because of Rule 8, no further subtransactions T'' ∈ext(T) can be initiated on C'.

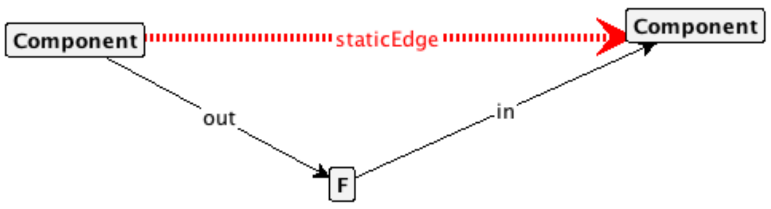
4. (past-validity) A past edge $C - \frac{past}{T} - > C'$ begins to exist no later than the end of any transaction $T' \in$ ext(T) initiated by a transaction hosted by C on C' through $C - \frac{static}{} - > C'$, and continues to exist at least until the end of T.

Proof. The proof is divided in two parts. The first part we prove that a past edge $C - \frac{past}{T} - > C'$ begins to exist no later than the end of any transaction $T' \in$ ext(T) initiated by a transaction hosted by C on C' through $C - \frac{static}{} - > C'$. This is true since the creation of past edges only occurs in Rule 10. When Rule 10 is applied, the termination of the first subtransaction on C' implies the contemporary creation of a past edge. The termination of future subtransactions is modeled in Rule 11 which, to be applied, requires the presence of the past edge. In the second part, we prove that a past edge $C - \frac{past}{T} - > C'$ continues to exist at least until the end of root transaction T. This is true since the removal of past edges occurs only during the cleanup step by applying Rule 15, and Rule 16. The application of these rules requires that the root transaction T is already terminated.

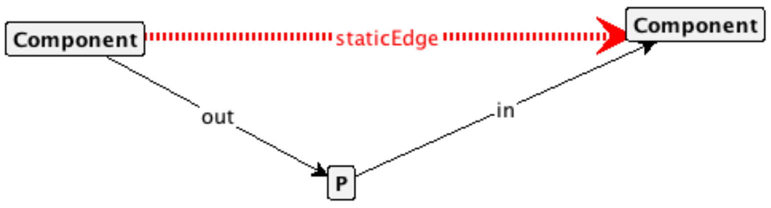
Model Checking

Locality:

LocalityViolationFuture



LocalityViolationPast

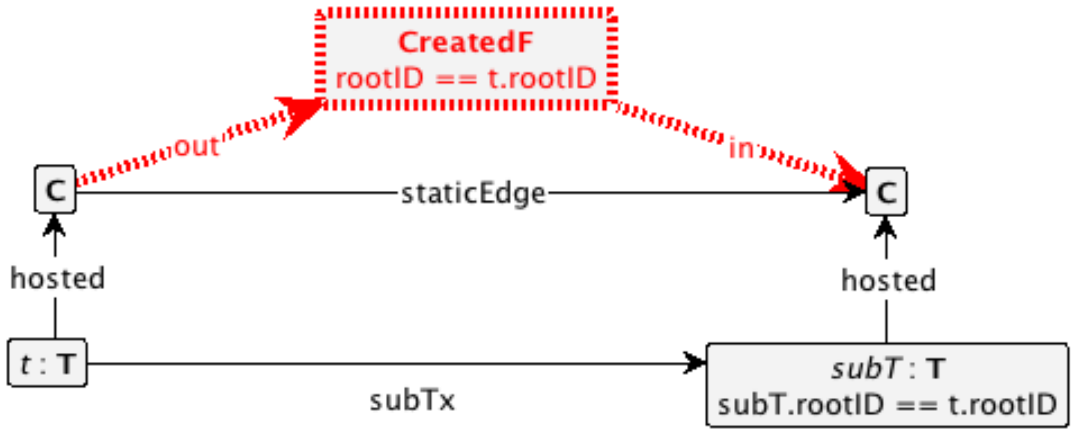


CTL formula: AG(!localityViolationPast & !localityViolationFuture)

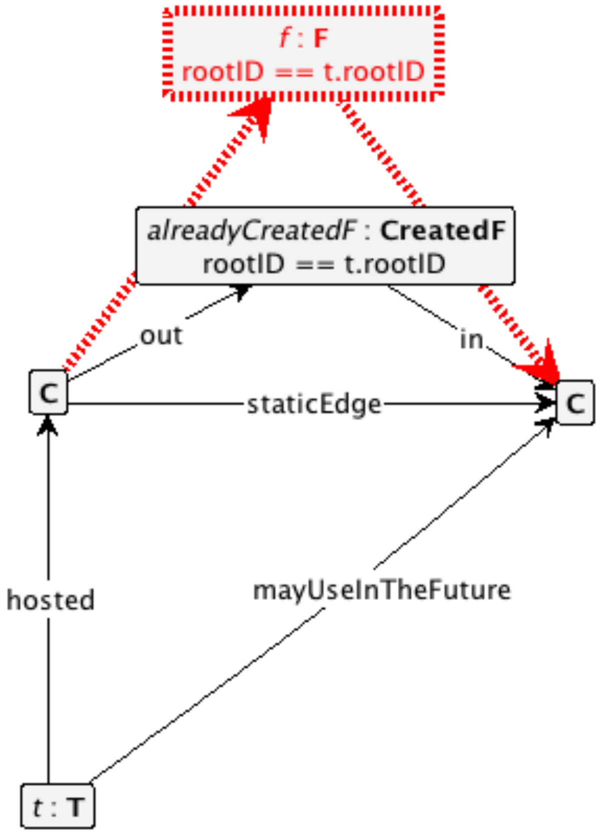
Description:

Along all paths starting from the initial state, and for all states no violation of the Locality occur both for future and past edge.

Future Validity:
FutureValidityViolation1: there exists a subTx **without** a future edge has been created.



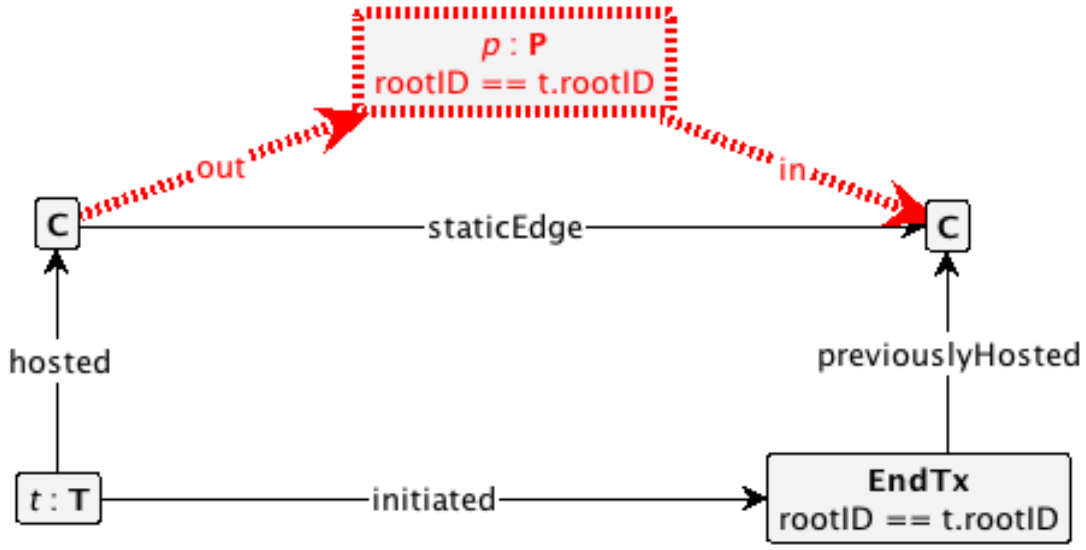
FutureValidityViolation2: if a component is hosting a transaction which mayUseInTheFuture another component **without a future edge.**



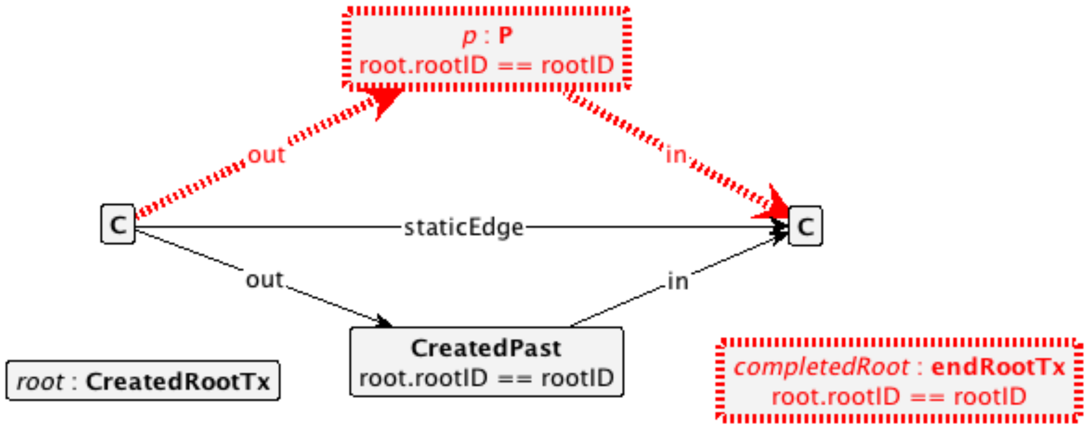
CTL Formula: AG(!futureValidityViolation2&!futureValidityViolation1)

Past Validity:

PastValidityViolation1: It is a violation if a transaction has ended without the creation of past edge.



PastValidityViolation2: it is a violation if a created past edge has been removed before the end of the associated root transaction.

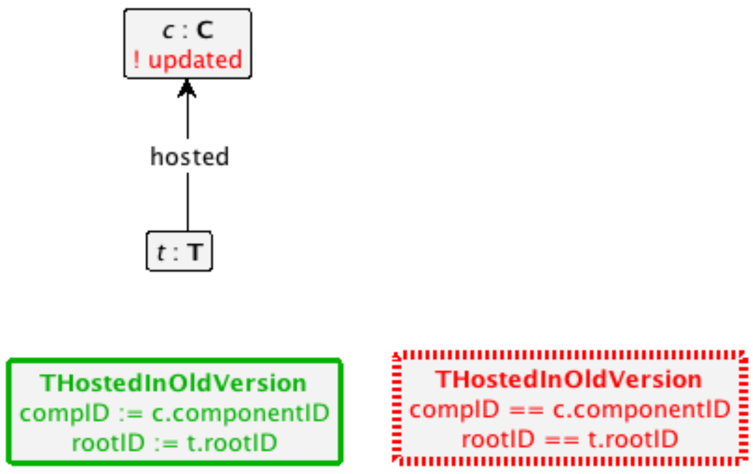


CTL Formula: AG(!pastValidityViolation1 & !pastValidityViolation2)

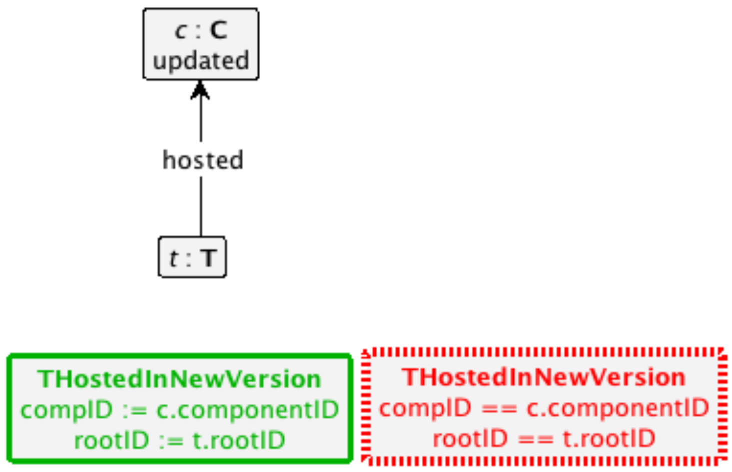
VERSION CONSISTENCY

We first need to remember the version of a given component used to host a given transaction.

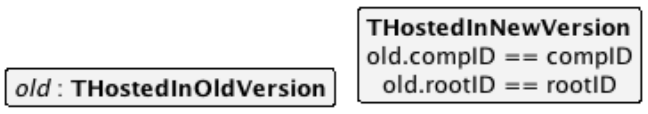
IdentifyOldComponentVersion



IdentifyNewComponentVersion



VersionConsistencyViolation: it is a violation of version consistency if transactions belonging to the same distributed transaction (with the same rootID) is hosted by different versions of the same component .



CTL Formula: AG(!versionConsistencyViolation)

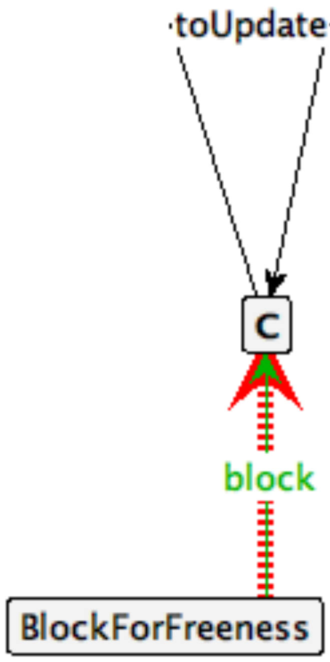
Formalizing the Strategies to Achieve Freeness

In this section we formalize the BF (Blocking for Freeness) and CV (Concurrent Version) strategies to achieve Freeness. The third strategy presented in the paper, namely WF (Waiting for Freeness), is implicitly formalized in the above sections.

1. Blocking for Freeness Formalization

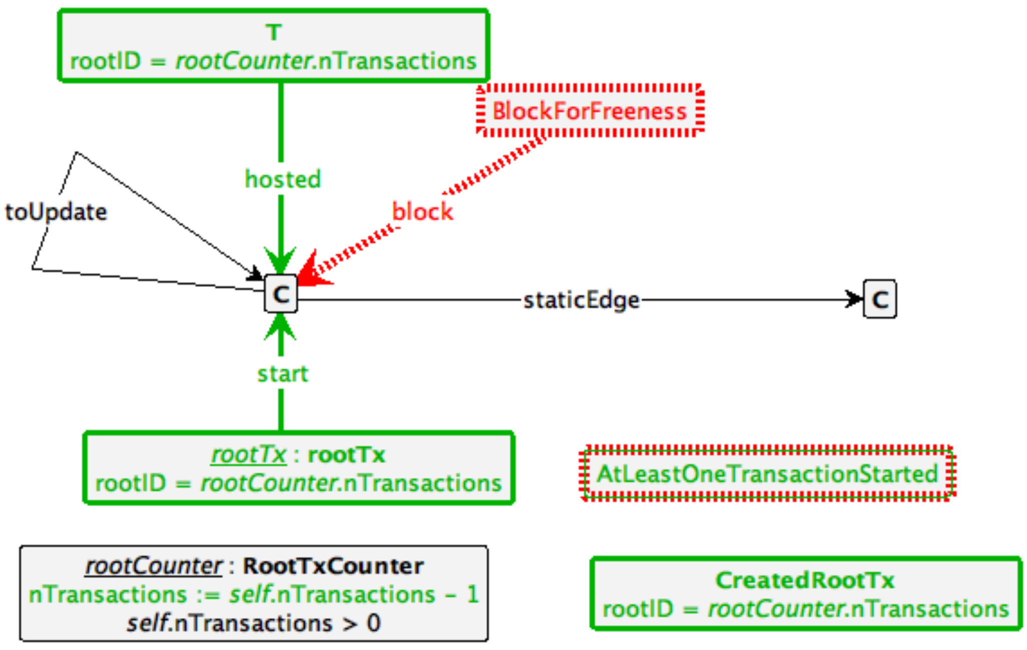
To formalize the BF strategy we add to the basic formalization the following rules.

1. **Start Block:**



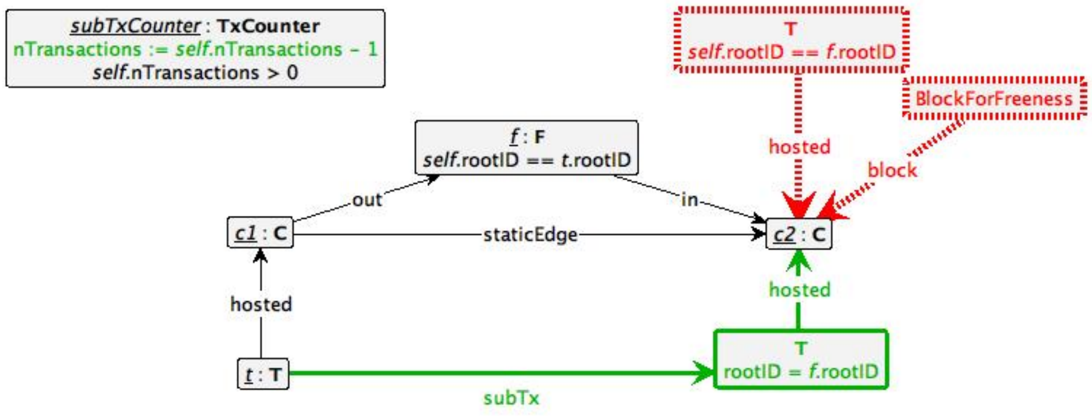
During the evolution of the algorithm, a block can be activated to the component to be updated.

2. StartRoot_To_Non_Blocked_Component


T
rootID = rootCounter.nTransactions
BlockForFreeness
hosted
block
toUpdate
C
staticEdge
C
start
rootTx : rootTx
rootID = rootCounter.nTransactions
AtLeastOneTransactionStarted
rootCounter : RootTxCounter
nTransactions := self.nTransactions - 1
self.nTransactions > 0
CreatedRootTx
rootID = rootCounter.nTransactions

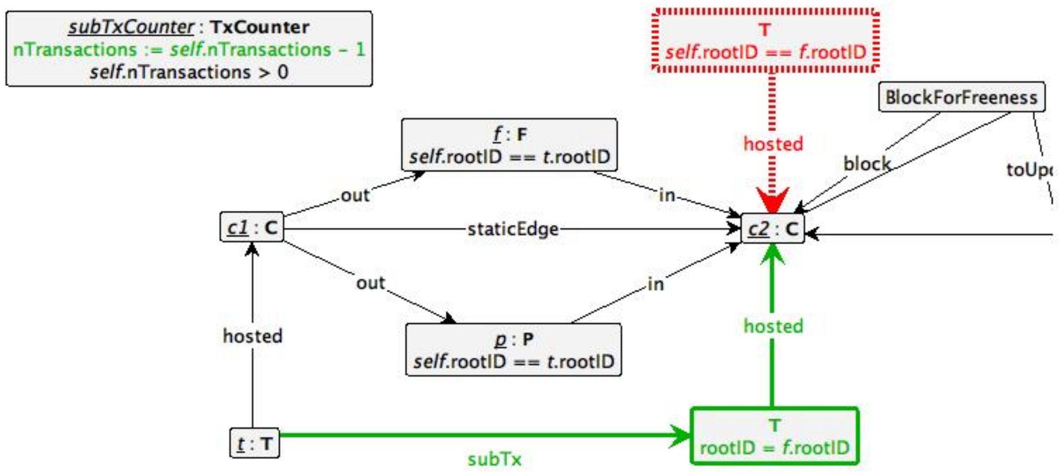
Root transactions can be initiated in any component. The component to be updated can initiate a transaction only if it is not blocked.

3. Start transaction when component is not blocked



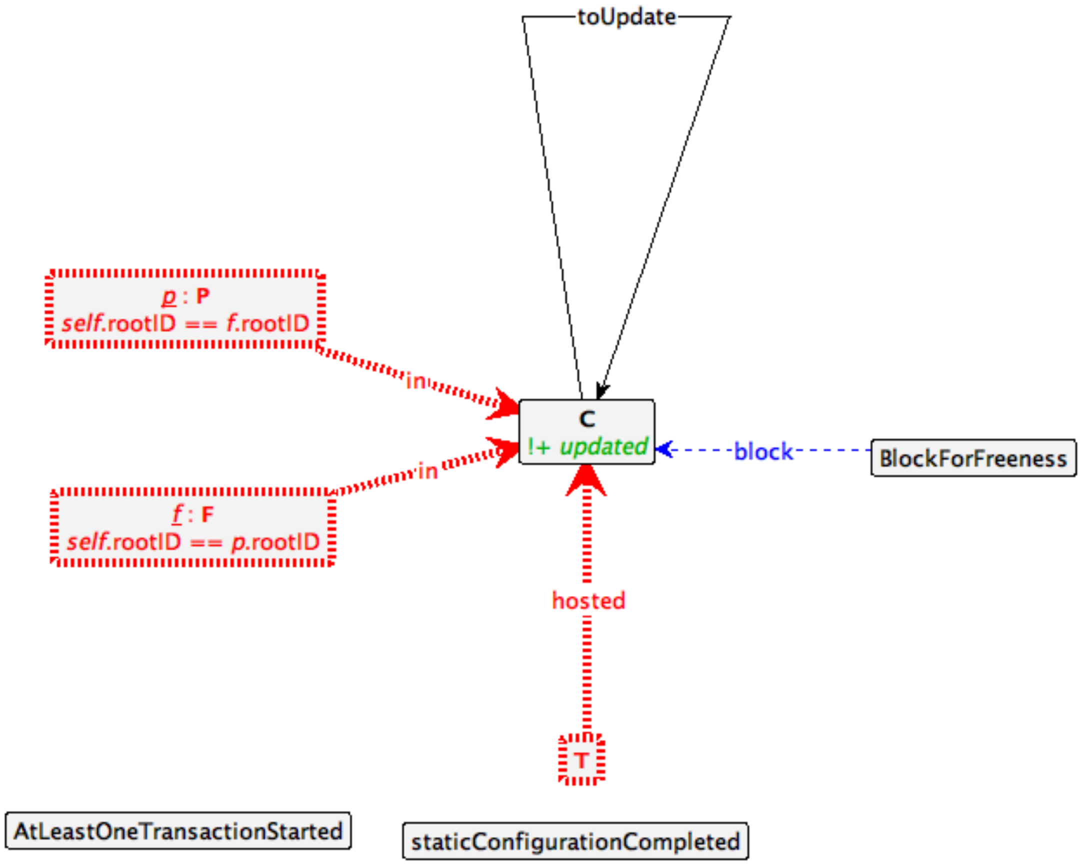
A sub-transaction can be initiated in a component to be updated if it is not blocked.

4. Start transaction when component is blocked



If the component to be updated is blocked, a sub-transaction can be initiated only when there is an incoming P with the same rootID.

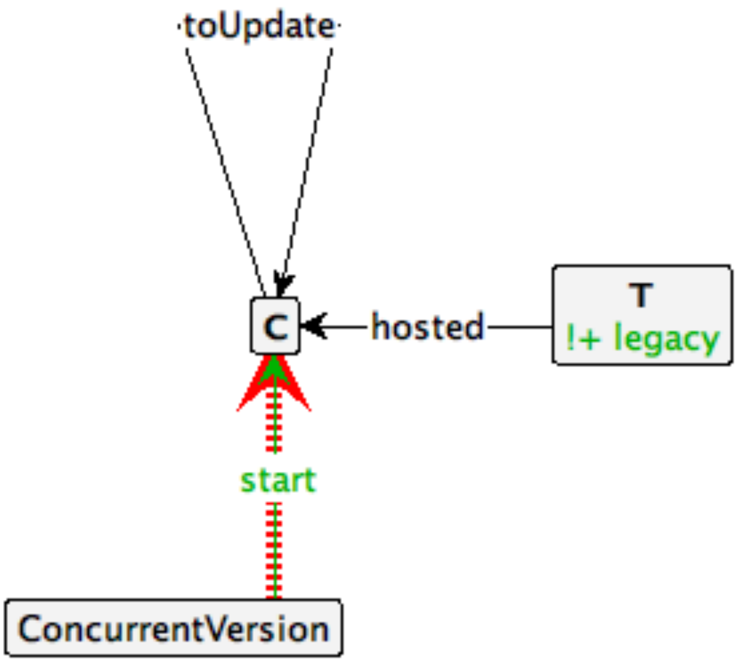
5. Remove Block

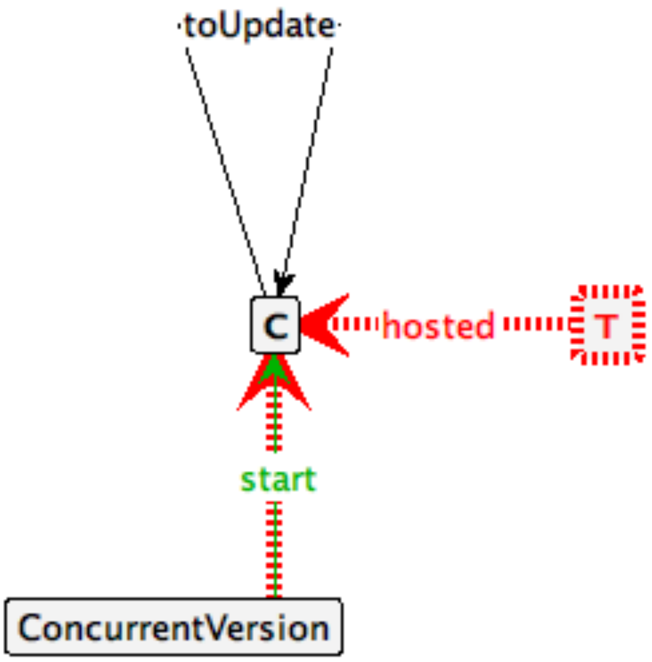


When the component is updated (i.e. when it reaches the freeness condition) the block is removed.

2. Concurrent Version Formalization

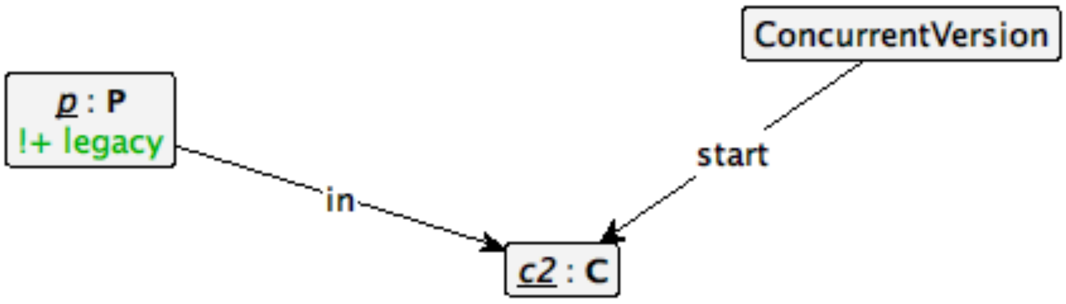
1. Start CV





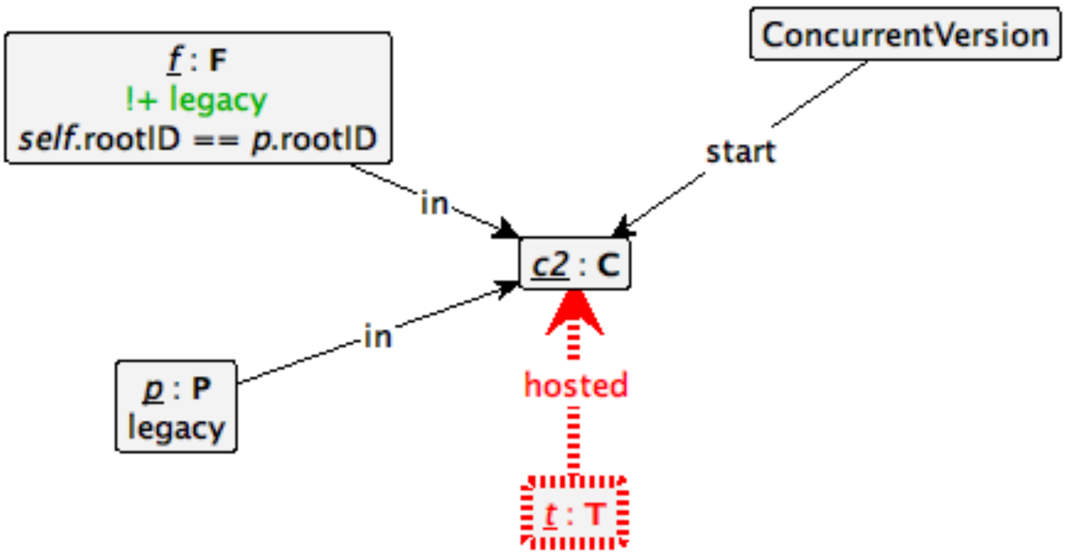
The CV strategy can start at any time. If the component is hosting a transaction, the transaction is marked as *legacy* to indicate that it is served by the old version.

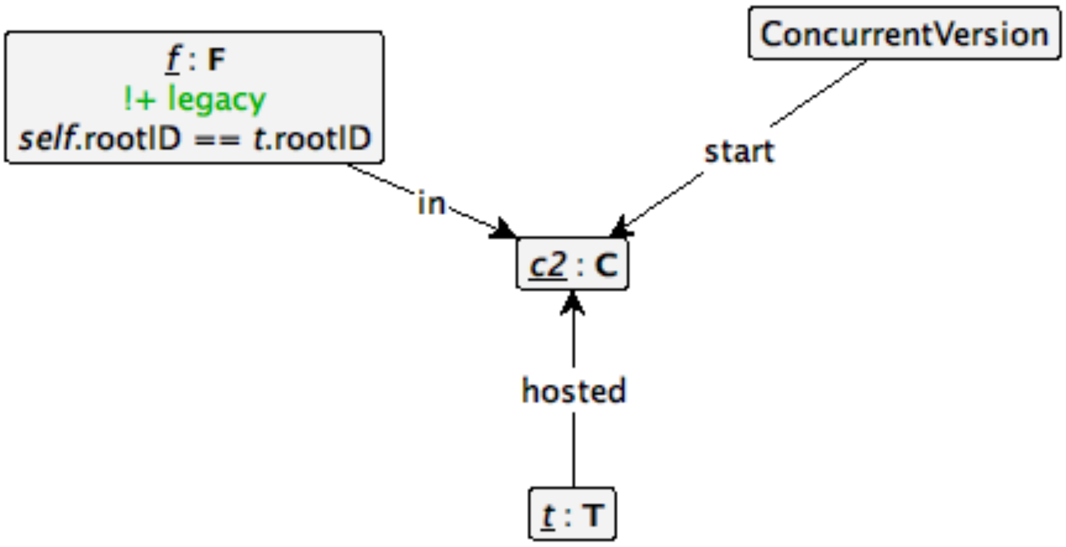
2. Tag Past Edges



If not tagged, tag an inward past edge to the target component with *legacy*.

3. Tag Future Edges

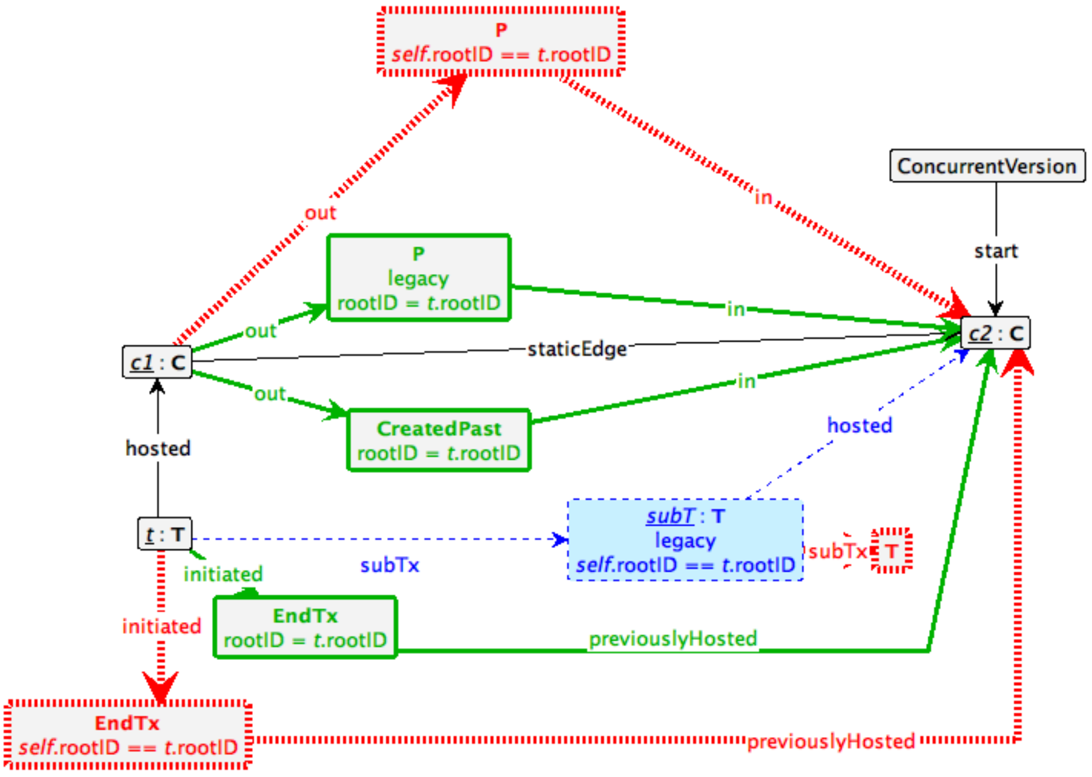




If not tagged, tag an inward future edge to the target component with *legacy* if there is a transaction with the same rootID running on the target component or there is an inward past edge with the same rootID;

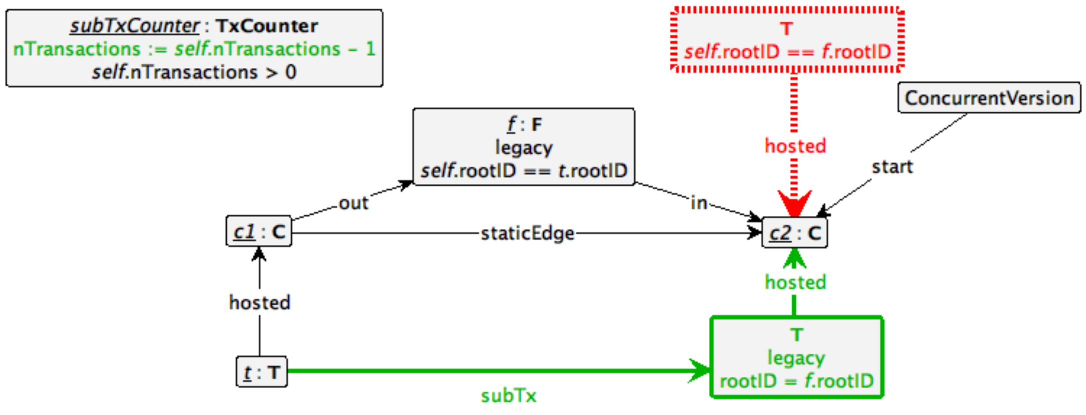
4. Creation of Past Edges

After CV activation stage, we add following higher priority rule corresponding to original rule for the creation of past edge.



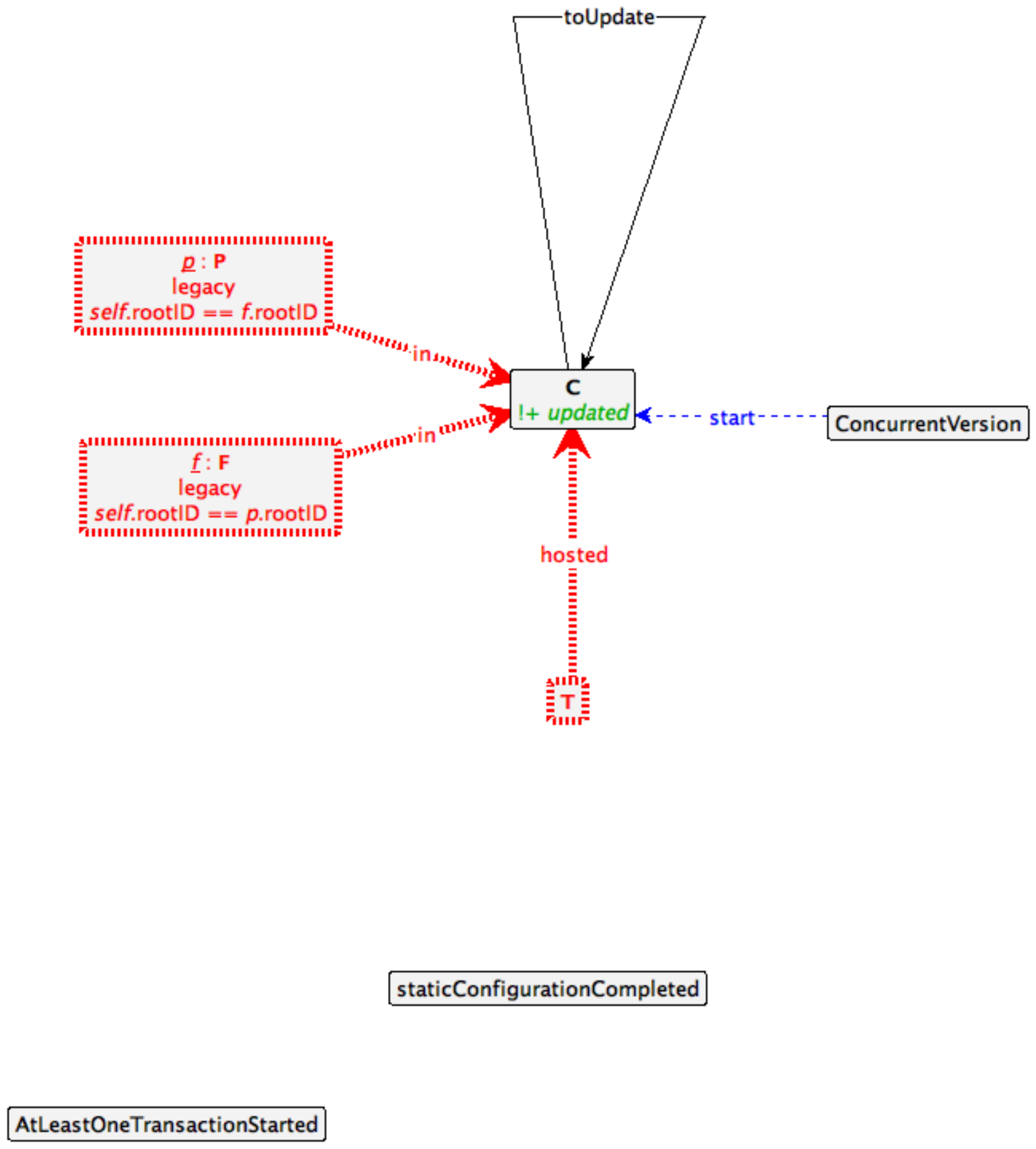
The creation of a past edge tagged with *legacy* just after the termination of the first sub-transaction subT tagged with Legacy initiated by c1 on c2. Note that original rule still takes effect, but at a lower priority for all those untagged elements.

5. Sub-transaction initiation



When a sub-transaction is initiated on the target component, the sub-transaction will be tagged as *legacy* if the sub-transaction is initiated under the condition that there is a *legacy* inward future edge with the same rootID.

6. Freeness condition

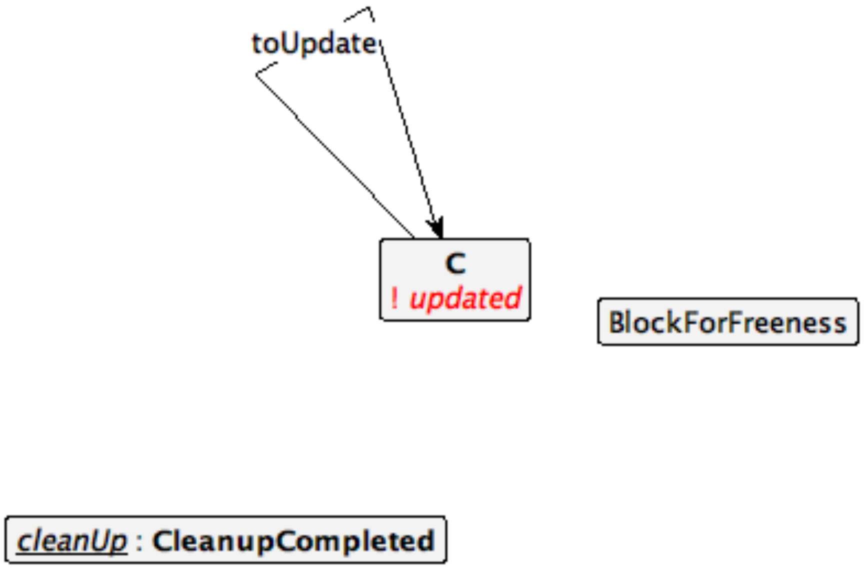


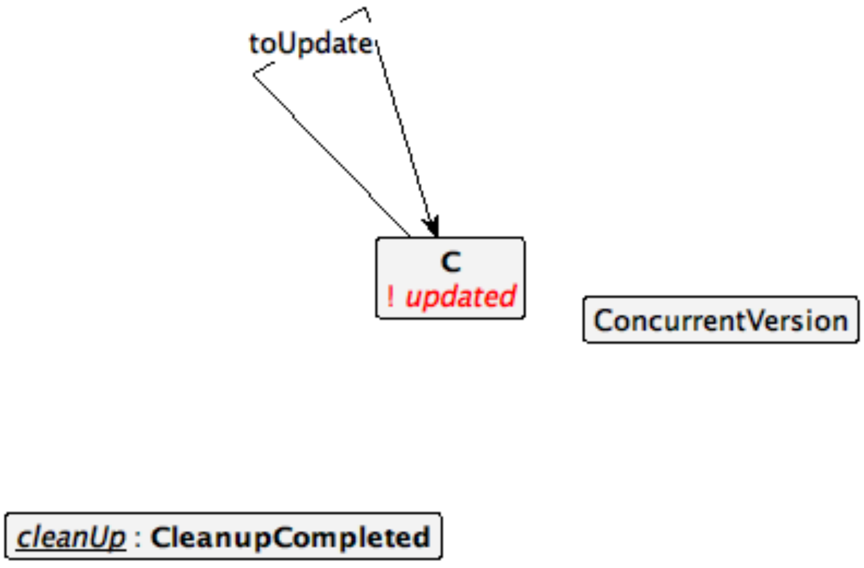
Rule about freeness is updated to consider the *legacy* tag.

3. Verifying liveness of BF and CV strategies

The proposed formalization is useful to use GROOVE to model check the liveness property of the two strategies. That is, we want to verify that, by applying one of the two rules, the freeness condition is eventually reached, and therefore the component can be safely updated.

To check this property we add the following condition rules (one for BF, and one for CV) and verify that it is not possible to reach the end of the algorithm without updating the component.
More in details we used the following CTL formulas:
AG(!checkUpdateViolation_CV)
AG(!checkUpdateViolation_BF)


toUpdate
C
! updated
BlockForFreeness
cleanUp : CleanupCompleted


toUpdate
C
! updated
ConcurrentVersion
cleanUp : CleanupCompleted