IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title: Consensus of History in Distributed DCR Graphs

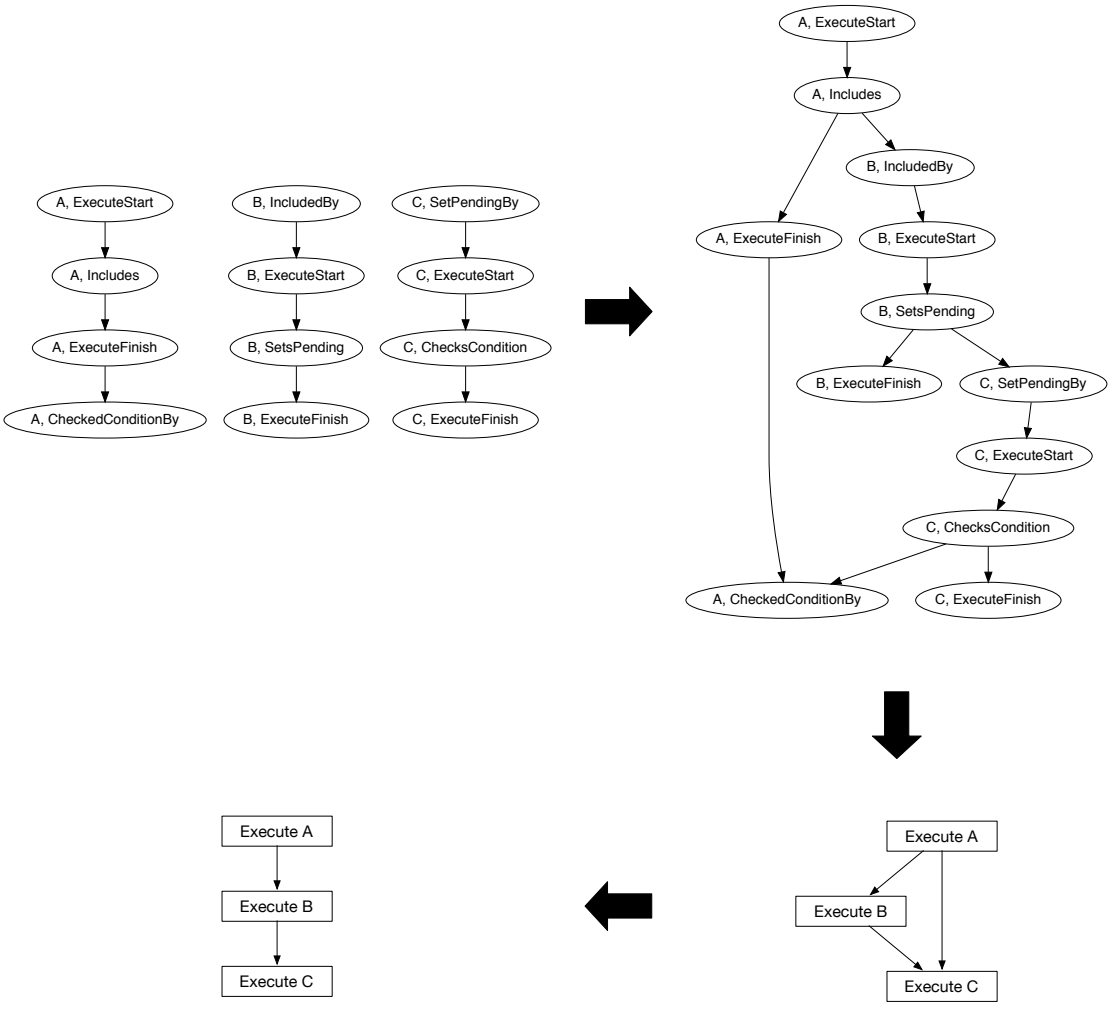
Supervisor: Søren Debois

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Anders Fischer-Nielsen	06/05-1993	afin@itu.dk
2.	Anders Wind Steffensen	10/02-1993	awis@itu.dk
3.	Mikael Lindemann Jepsen	17/02-1992	mlin@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Consensus of History in Distributed DCR Graphs

Anders Fischer-Nielsen
Anders Wind Steffensen
Mikael Lindemann Jepsen

May 17th 2016



Contents

Chapter 1

Introduction

Dynamic Condition Response Graphs (DCR graphs, see [8]) are developed principally by Thomas Hildebrandt at the IT University of Copenhagen, in collaboration with ResultMaker[1] and later Exformatics A/S[2]. DCR graphs are used to model workflows which represents work processes.

DCR graphs are used by companies to ensure that business procedures are executed according to business rules. At a given time, a business might want to see in which order events in a given DCR graph have executed, a so called *order of execution*. The order of execution can contain interesting information for the business, as it reveals which events have been executed to lead to the current state of the workflow. Furthermore, if a DCR graph is used to represent a case of a customer, and if multiple case workers are working on that case, and do not have a total overview of it, an order of execution can provide that overview.

Since DCR graphs allow infinite behaviour, the workflow can be in the same state multiple times. This leaves the case worker guessing how many times the workflow has been in the current state. Finally, for DCR graphs shared between two or more companies, one of the participating companies might want to check retroactively whether the other companies have followed the rules of the workflow.

The events of a DCR graph can be distributed on a number of processes over a network allowing better scalability, reliability as well as allowing multiple companies to work together, by hosting and handling different events themselves.

In a distributed DCR graph finding an order of execution can be difficult, because no single process has an overview over the entire workflow. Information about what has happened can be split among several processes, the clocks of processes are not necessarily synchronised, and processes may act maliciously, that is they may not follow the rules of the workflow or they may emit erroneous information. Even though there are challenges, DCR graphs provide information that can be used to relate executions with one another, which is especially helpful when information of two or more events need to be combined into an order of execution.

Because there exists no central source of truth in a non-centralized distributed system, the processes must reach consensus on the work they have done together, for anyone to believe that the work is correct.

[1] `http://www.resultmaker.com`
[2] `http://www.exformatics.dk`

Any malicious activity should be observable and if possible the events that are behind this activity should be identified. Handling message tampering, that is altering, replaying and withholding messages before sending them to the intended recipient as described in [1], and processes crashing while carrying out a task, are out of scope for this project.

By researching and applying distributed system theories and algorithms, as well as the rules of DCR graphs, we will try to solve the problem of generating and reaching consensus of histories of DCR graphs.

This leads us to the problem definition.

1.1 Problem Definition

Given an execution of a distributed DCR graph, the purpose of this project is to find an algorithm which uses the local orders of execution of the individual events to find a global order of execution with the least amount of concurrency, where the local orders are preserved in the global order. Furthermore, the algorithm should be able to observe and if possible, identify which events in the workflow are part of malicious behaviour, where malicious behaviour is the act of disobeying the rules of the workflow or reporting false information. Finally, the algorithm should establish distributed consensus on the order of execution among the events.

This report has the following structure:

Chapter 2 describes the previous work that this project is based upon. This includes distributed DCR graphs, graph theory, as well as distributed system concepts, such as consensus algorithms, serial equivalence and ordering of events.

Chapter 3 introduces the terms action and history and defines the representation of a history and an action. Furthermore, it examines why a local history of a single event must be in a strict total order and why this property is necessary when finding ordered history of multiple events.

Chapter 4 examines how it is possible to exploit the rules of DCR graphs when merging histories of multiple events into a single history and explains why the result is in strict partial order. A valid history is defined and it is assumed that all histories adhere this definition.

Chapter 5 examines how it is possible to find an order of execution from a global history, and to prove the correctness of that order of execution it is shown that the participating events of the history can agree on the result.

Chapter 6 introduces malicious processes in the implementation of DCR graphs and what such processes can do to a history. Furthermore, it describes how it is possible, both locally and globally, to validate histories using serial equivalence, ordering of events, and the constraints of DCR graphs.

Chapter 2

Background

2.1 Dynamic Condition Response Graphs

A Dynamic Condition Response Graph is a declarative, event-based process model. Instead of using sequences of state transitions like imperative process languages, declarative process languages uses sets of constraints to model the possible transitions in the process (hereafter *workflow*). A benefit of DCR graphs is that they allow finite specifications of infinite behaviour.

A DCR graph is in [8] defined as a tuple $(E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ where

- E is the set of events
- M is the marking, the state of the workflow. The marking is a triple containing three sets of events. The first set contains the events that have previously been executed. The second set contains the events that are required to be executed or excluded. These are called the pending responses. The last set contains the events which are currently included.
- Act is the set of *actions*. That is, a representation of what happens in the workflow when a given event is executed.
- $\rightarrow\bullet$(conditions) is a relation, which for all pairs of events which are present in the relation, means that the first event must be either excluded or executed for the second event to be executable.
- $\bullet\rightarrow$(responses) is a relation, which for all pairs of events which are present in the relation, means that after the first event has been executed the second event must either be executed or excluded at some point for the workflow to be in an accepting state.
- $\pm$ defines the dynamic inclusion and exclusion of events. This partial function is a triple of two events and symbol defining whether the execution of the first event should result in an exclusion or inclusion of the second event. It is possible for an event to exclude itself, but one event cannot both include and exclude another.
- l is the labelling function, assigning an *action* to each event.

Furthermore the article defines a distributed DCR graph as a DCR graph with a set of roles, a set of principals, and a mapping function that assigns roles to principals and actions. This

assignment means that if principal P is assigned role R, and action A is assigned R, then P can execute A.

Events can be safely distributed by defining *projections* and *compositions* of DCR graphs. This is described in [7]. This idea of distributing events between multiple processes in a distributed system is used in the DCR graph implementation used for this project. This implementation is further described in section 2.4.

In [5], Debois et al. describes the procedure for executing distributed events using a locking mechanism to ensure serial equivalence:

> "*The procedure for executing an event, in detail, is as follows. A component wishing to execute an event e must first request*[1] *and receive locks on all (local and remote) events that are in conflict (i.e., not independent) with e (thus, in particular, on itself). It then queries the state of remote events to determine if e is currently executable. If it is, it instructs remote events affected by firing e to change state accordingly. Finally, it releases all locks.*"

Furthermore, Debois et al. states in [5] that for two events to have concurrent executions the events must be independent, that is the events must be effect and cause-orthogonal. In the article effect-orthogonality is described as

- "no event included by e is excluded by f and vice versa, and"
- "e requires a response from some g iff f does."

and cause-orthogonality is described as

- "neither event is a condition for the other,"
- "neither event includes or excludes the other, and"
- "neither event includes or excludes a condition of the other."

2.2 Distributed Systems

Distributed systems is an area of computer science which focuses on the concepts, problems and design of computer systems where message transmission among processes are not negligible compared to the time between events in a single process.[2] Typically these systems are distributed on computers in a network.

2.2.1 Serial Equivalence

Operations performed by two concurrent processes are serially equivalent if the resulting system state is the same as if any one of the processes' operations happened first, followed by the other. In distributed systems this term is used for transactions across machines in a network and is especially important since message passing over network connections introduce inconsistent and unpredictable delays. Therefore multiple messages can arrive at different times, even if they were supposed to happen in sequence. An example of a serial equivalent interleaving of two

[1] All components request locks in the same fixed order to prevent deadlocks.
[2] Lamport describes distributed systems as this in [9]

transactions is shown in figure 2.1. Common implementations of achieving serial equivalence in distributed systems include locking and optimistic concurrency control. In [2] these concepts are explored in depth.

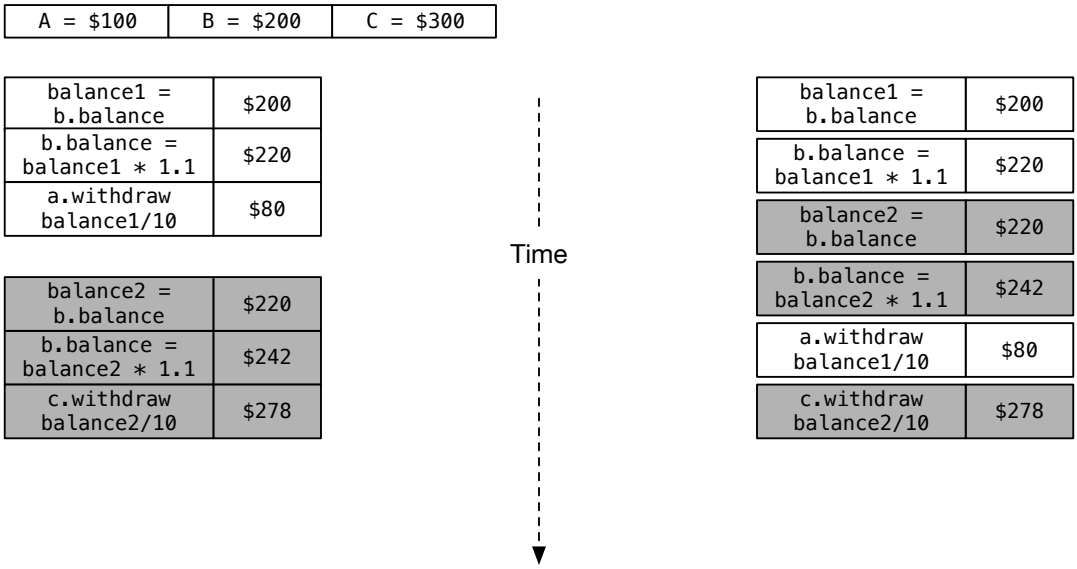


Figure 2.1: Two transactions executed in sequence and a serially equivalent interleaving.

2.2.2 Ordering of Events

In [9] Leslie Lamport examines and describes the ordering of events in distributed systems (not to be confused with DCR events) based on their occurrence in time.

In set theory a strict total order is an order that is *antisymmetric*, that is if $a \leq b$ and $b \leq a$ then $a = b$, *transitive*, that is if $a \leq b$ and $b \leq c$ then $a \leq c$, *total*, that is $a \leq b$ or $b \leq a$, in addition to being *irreflexive*, that is $a \not< a$. A *strict partial order* is similar to a *strict total order*, but does not have the *totality* property.

These orders can be described with two examples, using a subset of the natural numbers, $\{1, 2, 3, 4, 5, 6\}$ referred to as N. An example of a strict total order is N ordered by the less than ($<$) relation. An example of a strict partial order is N ordered by the proper divisors. An illustration of these two orders can be found seen on figure 2.2 and figure 2.3.

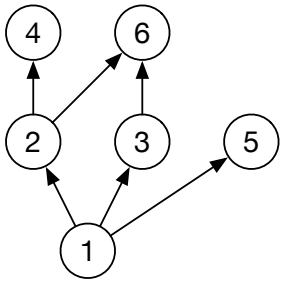


Figure 2.2: The strict partial order of N ordered by the proper divisors. $A \to B$ denotes A is a proper divisor of B.

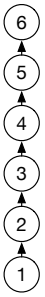


Figure 2.3: The strict total order of N ordered by the less than ($<$) relation.

Lamport introduces so-called *logical clocks*, which are simple timestamps that enable determining the order of events in a distributed system. The clocks are *logical* because they do not represent *physical* time, since synchronising physical time in a distributed system is a non-trivial task. A logical clock is a simple counter that is incremented for each event happening in a process.

When a process receives a message, the counter must be set to a higher value than both the received timestamp, as well as the local clock. This also implies that each message must include the timestamp of the sending event. This is illustrated in figure 2.4

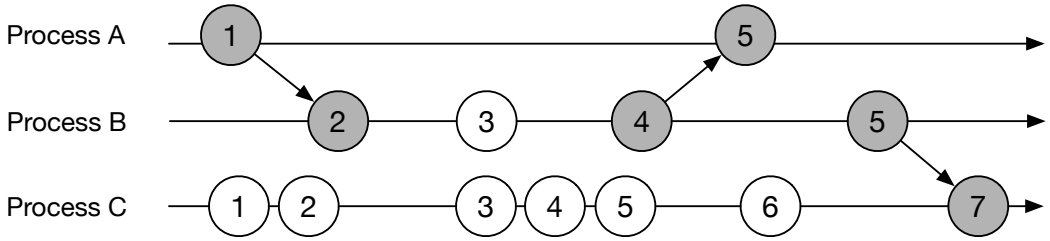


Figure 2.4: Three processes exchanging messages and setting their local timestamps accordingly.

Lamport argues that if a given system is not distributed it is possible to create a history of events where the ordering is total, since it is possible for one process to use a logical clock to find out which event happened before another. In a distributed system however, this global ordering is not possible since events have independent locical clocks that are set to the same timestamp.

Lamport describes these relations between events with the symbol $\to$ such that if *event* a happens before *event* b then $a \to b$. These relations can be created based on three cases:

1. If *event* a and b happened on the same process and a happened before b measured with the local logical clock then $a \to b$.
2. If process i sends a message to process j and *event* a is the event which initiates the sending and *event* b is the recieving of the message then $a \to b$.

3. If $a \to b$ and $b \to c$ then $a \to c$, that is, it is transitive.

If no relation exists between any event a and b, the events are *concurrent* and logically occur at the same time, even though this might not be the case physically. Lamport therefore argues that it is only possible to find a strict partial order of events in distributed systems.

2.2.3 Consensus

Traditionally consensus in distributed systems is the activity of reaching agreement among the participants in the system on a proposed value. The consensus problem has the following three requirements as stated in [3]:

- Termination: Eventually each correct process sets its decision variable.
- Agreement: The decision values of all correct processes are the same.
- Integrity: If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

Such problems are often achieved by using an consensus algorithm such as Paxos [10] and Raft [12].

Assumptions about the distributed system are made for both Paxos and Raft, namely, processes operate at an arbitrary speed, may experience failures, may recover after a crash by storing their state, and do not collude, lie or attempt to subvert the protocol; in short, Byzantine failures do not occur.

In our problem domain processes may operate at an arbitrary speed and *may* collude, lie or attempt to subvert the protocol; Byzantine failures *do* occur. Therefore it is not possible to use Paxos or Raft for this project.

The *Byzantine generals problem* is a case of the consensus problem. Instead of having multiple proposers, the commander is introduced. Byzantine failures are allowed and handled by signing messages. Also, each process needs to be connected to all other processes and there is a limit of a third malicious processes that can exist in the system. It has been proven that the Byzantine generals problem cannot be solved for a system with only two processes. [11]

2.3 Directed Graphs

A *directed graph* is a data structure which consists of *nodes* connected by *edges*. In directed graphs an edge is an ordered pair of nodes, where the first element is the start node and the second is the end node. When traversing the graph, it is only possible to traverse edges from their start node to their end node. There exists a *path* from a node to another if it is possible through one or more edges in succession to get from the first to the second node.

For a given node, the *reachability* is the set of all the nodes there exists a path to. For a graph to be *acyclic* any node in the graph must not have a path to itself.

The subgraph of a given graph with the same reachability of each node, but with as few edges as possible, is called the *minimum equivalent graph*.

A *topological order* of a directed graph is a total ordering of the nodes, such that for every edge a, b from node a to b, a comes before b.

All of these topics are described further in [13].

2.4 Distributed DCR implementation

We have based the implementation of this project on the work done during our second year project.[3]

From the requirements of the second year project:

> "*The goal is to develop a functioning and correct, web-service based distributed workflow system that can support distributed coordination of workflows provided by the external and possibly international customers, and reconfigured if the workflow changes.*"

At the time of implementation, only *condition*, *response*, *inclusion*, and *exclusion* relations were required, even though other relations exist, such as the *milestone* and *spawn* relations.

In order to fulfil the requirements we separated the project into smaller components:

- An event *peer*
- A central *server*
- A *client*
- A parser and uploader to set up the workflows

An event peer can contain zero or more DCR events. For each of these events, the peer persists information about the event. This information consists of an ID, a name to show in the client in addition to the state of the event which is represented as boolean values for include, pending, and execute. This corresponds to the marking of the projection of a DCR graph where only a single event is chosen as projection parameter. The relations of an event are also stored. Furthermore the initial values of the state are stored in order to be able to restore the workflow to the initial state.

As mentioned above and as opposed to the definition of DCR graphs in [8] the states of workflows are not saved as markings. Instead the event peer saves the state of each of the events it hosts.

In order to achieve serial equivalence when executing, a locking strategy has been used because of its simplicity and because performance was not a priority for the project. The system executes events using the procedure from [5] as quoted in section 2.1. In order to prevent deadlocks, the order of obtaining locks has been defined inside a workflow. Locks are acquired according to alphabetical order of the IDs of events.

The central server contains information about workflows and the events that take part in the workflow. Each workflow is identified by an ID, that is chosen when creating the workflow. For each event in a workflow the ID of the event as well as the address of the event peer hosting that event is stored. The server also contains information about users and the roles they are operating with.

[3] For an in depth description of the original implementation the project including a report can be found at `https://github.com/andersfischernielsen/FlowIT-Second-Year-Project/`.

In the client, when a user logs in, the server sends information about which roles the user has access to in the workflows he is connected to. The client can then filter the events of a chosen workflow, so the user only sees events where he or she has the right to execute.

For each accessible event in a workflow, the client shows the name and state of the event. Furthermore, a button used to execute an event exists, which is deactivated if the event is not executable. The client also allows the user to get an overview of what happened in a given workflow. This feature was called "history", but was implemented by ordering each log entry by physical time. This history feature has obviously been replaced with the outcome of this project.

The parser is a tool developed to make it easier to upload workflows created on DCRGraphs.net[4] into the system.

2.4.1 Changes to the original implementation

When starting this project, we discovered a few mistakes in the original implementation. Most importantly, conditions were checked before locking the target events, which could mean that this conditioning event could change its state and therefore change whether or not the condition was fulfilled, before the locks on other events were obtained.

Furthermore the aforementioned history implementation has been removed and the parser is now included in the main client project.

Increasing timestamps for messages received and sent between events are also implemented in ordert to use Lamport timestamps. Timestamps between events should always increase, which means that an event will reject a message with a timestamp lower than the previous timestamp it has received from another event.

In order to store any kind of information on an event, it must be locked. Furthermore, information about locks are not stored on events.

In the aforementioned report, the user guide points to certain URLs that should contain the web services of the system. These web services are no longer running.

[4] `http://dcrgraphs.net`

Chapter 3

Representing a History

This chapter defines the representation of a *history*. Furthermore it examines why a local history of a single event is a strict total order and a history of a workflow is a strict partial order.

3.1 Representation of an Execution

In order to find the order of execution in a workflow, one must first determine how to represent an execution. When an event executes it affects all of its relations. Furthermore since executions can happen multiple times, an execution also exists at a certain point in time, and so does every effect. This leads to the first definition of an ***execution***:

Definition 1. *An **execution** of an event can be represented as a finite set of effects, and their timestamps. The set of effects are determined by the rules of the workflow.*

Timestamps based on system clocks in distributed systems are not always synchronized, as described in section 2.2.2, and therefore Lamport's logical clocks can be used to determine the place in time of each effect. Since an effect both has an event which performs the effect and another event which gets affected, we say that each effect has a ***counterpart***[1]:

Definition 2. *In a given effect where two events A and B are part of the effect, event B is **counterpart** of event A and event A is **counterpart** of event B.*

To distinguish between the two events which takes part in the effect, it is necessary to use the IDs of both. Furthermore, to distinguish effects from each other an effect type is necessary. Since an execution can happen multiple times and therefore affect the same counterpart multiple times, the counterpart timestamp is also required to distinguish different effects on the counterpart from each other. To distinguish multiple executions from each other, an execution must have a start time and end time.

For an event to know that something has happened, it is necessary to store it for later retrieval. This must be done for all effects on both sides of the effect.

To make a common representation for what is stored, when an effect happens or the execution starts and finishes, the ***action*** is introduced:

[1] The idea of executions having effects on counterparts has been explored in [6]

Definition 3. *An **action** consists of the ID of an event, the time stamp of the action, the ID and timestamp of a counterpart if relevant, as well as the type of the action.*

The type of an action is defined as an ***action type***:

Definition 4. *An **action type** can be one of the following:*

1. *Checks condition*
2. *Checked condition by*
3. *Excludes*
4. *Excluded by*
5. *Includes*
6. *Included by*
7. *Sets pending*
8. *Set pending by*
9. *Execute start*
10. *Execute finish*

The action types 1 through 8 are related to the functionality of DCR graphs, i.e. *Includes* is an action that happens when an event A includes event B, similarly *Included by* is an action that happens on event B when event A includes it. Action types 9 and 10 happen when an event starts executing and finishes executing.

Actions can form pairs according to their action types. Action types 1 through 8 in definition 4 represent the types of the two sides of the effect.

- *Outgoing action types* are action types 1, 3, 5, and 7.
- *Incoming action types* are action types 2, 4, 6, and 8.

For readability an *outgoing action* is an action with an outgoing action type. In the same fashion an *incoming action* is an action with an incoming action type.

In this report we will use the following illustration types for signifying different entities:

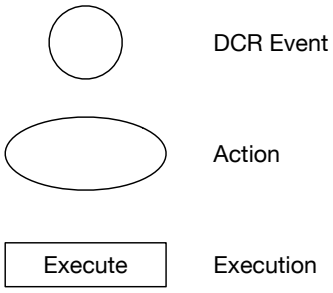


Figure 3.1: A legend showing what the different figures mean in the following sections.

An execution contains all the actions that was created as the result of the execution of an event, as illustrated in figure 3.2.

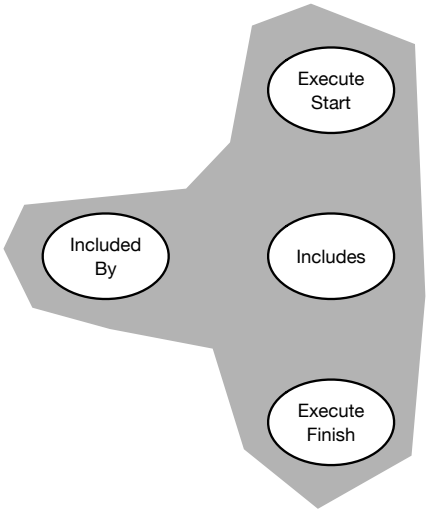


Figure 3.2: Actions that are part of an execution.

With these new informations the definition of a representation of an ***exeuction*** can be revised to:

Definition 5. *An **execution** of an event can be represented as a finite set of actions, which represents the effects that happened in the execution including two actions with action types **Execute start** and **Execute finish**, respectively.*

The ingoing actions of an execution will be stored at the counterpart which is not always the same event, as the one executing. This situation can be seen on figure 3.3.

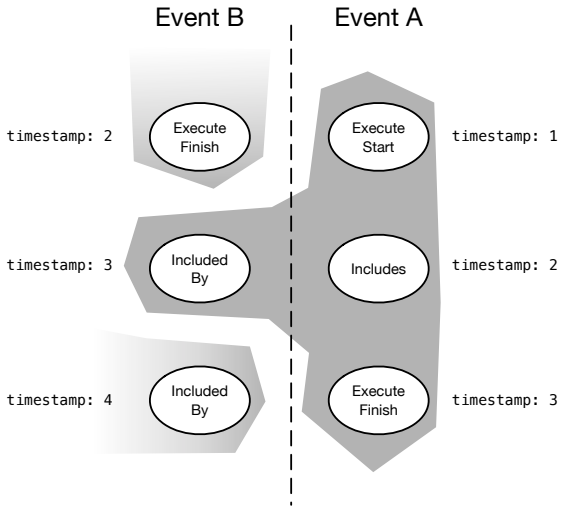


Figure 3.3: Actions that are part of an execution where an ingoing action is stored on another event.

3.2 Representation of a History

Now that a representation of an execution and its actions has been defined, it is necessary to find a way to connect and order these actions.

When looking solely at the actions of a single event, we can compare each action with any other action and see which happened first by comparing their timestamps. For an action to be part of an execution, it has to happen after an *Execute start* and before an *Execute finish*. Due to the requirement of DCR graphs that executions in an implementation must happen in a serially equivalent manner, all actions in execution A must either happen before or after all actions in execution B if the two executions are executions of the same event. Therefore, executions and their actions happening on the same event are in *strict total order*. Recall that Lamport states that it is not always possible to establish a strict total order of events in a distributed system, as described in section 2.2.2. Therefore it is necessary to describe an invariant for executions of multiple events in a workflow:

$$\begin{gathered}(\exists x \in E_A, y \in E_B(x \to y) \implies \forall x \in E_A, y \in E_B(y \not\to x)) \\ \wedge \\ \neg(\exists x \in E_A, y \in E_B(x \to y)) \implies \forall x \in E_A, y \in E_B(x \not\to y \wedge y \not\to x)\end{gathered}$$

Invariant 1: Invariant for the actions of executions in histories

Invariant 1 reads, if there exists an action, x, in execution A, and another action, y, in execution B, such that x happens before y, then for all actions x in A and all actions y in B, y must not happen before x. Otherwise there are no happens before relations between actions in A and actions in B.

One can easily argue that a strict partial order fits ordering of actions because:

1. An action A cannot happen before itself, therefore the order must be *irreflexive*.
2. Actions A and B cannot both be happening before and after each other, therefore the order must be *anti-symmetric*.
3. If action A happened before action B and B happened before C then A happened before C and the order must therefore be *transitive*.
4. Actions can in distributed systems be concurrent and the order can therefore not be *total*.

This leads to the following definition of ***history***:

Definition 6. *A **history** is a strict partial order of actions.*

A *global history* is the strict partial order of the actions of all events in the workflow. A *local history* is a strict total order of the actions that have happened on a single event.

Since a finite strict partial order can be represented as a directed acyclic graph, the history can therefore also be represented as such, where actions are nodes and the order in which the actions have happened is represented as edges.

Chapter 4

Connecting Histories

In this chapter, the representation of histories is used to combine multiple local histories into one global history using the information of actions. Furthermore two approaches to gather histories from the events are described.

4.1 Merging Histories

Given the representation of a history, and because the actions of executions are distributed among several events, we need a way to connect the actions by happens-before relations.

In order to simplify this problem, it is assumed that all local histories are ***valid***. That is:

Definition 7. *A **valid history** is a history, where all actions happen according to the rules of the DCR graph, abiding serial equivalence and being in strict partial order. Also, a valid history **must** contain exactly the actions that **did** happen in the execution of events.*

The observation and identification of invalid histories are described in chapter 6. Why it is necessary to have this assumption is discussed in the end of this section.

Given valid local histories of several events, we want to connect the actions of the histories such that the result has the fewest possible concurrent actions using happens-before relations. Furthermore, given a global history A and every local history B used for the creation of A, for every path from action x to another action y in B, there must also exist a path from x to y in A.

The concept of happens-before relations, as described in section 2.2.2 helps determining which actions have happened before others across events. For each outgoing action in a history, there must be another incoming action, because these actions are the two sides of the same effect. Therefore, each outgoing action type corresponds to an incoming action type by the following definition:

Definition 8. *The action types that **correspond** are:*

- *Checks condition → Checked condition by*
- *Excludes → Excluded by*

- *Includes* → *Included by*
- *Sets pending* → *Set pending by*

For any outgoing or incoming action on an event, there might exist any number of actions with corresponding action types. Actions should therefore be matched on more than just their corresponding action type. An action includes the ID and timestamp of both the executing and counterpart event to ensure that matches are unique. This brings us to the definition of a **match**:

Definition 9. *A pair of actions, a and b,* ***match*** *if and only if they have corresponding action types and the ID and timestamp of a is identical with the counterpart ID and counterpart timestamp of b. Furthermore, the ID and timestamp of b must be identical with the counterpart ID and counterpart timestamp of a.*

Figure 4.1 is an example of a pair of matching actions. In figure 4.2 the actions do not match.

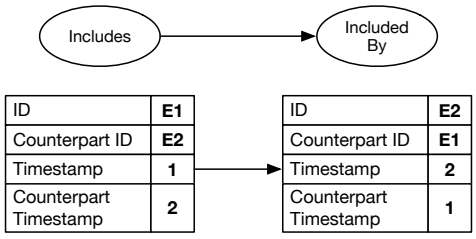


Figure 4.1: Two actions that match.

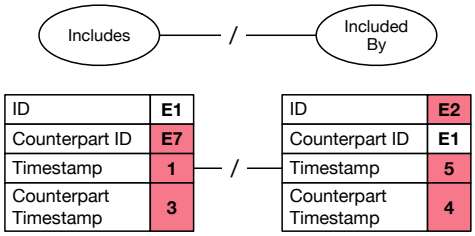


Figure 4.2: Two actions that do not match. Fields that do not match are marked with red.

An outgoing action must happen before its incoming counterpart, because together they represent a message exchange between to processes. A happens-before relation is established between the matching actions represented by adding an edge from the outgoing action to the incoming action in the history.

To create a history of the entire set of events, every action of every local history needs to be added to a joined global history. Therefore the algorithm must take a set of local histories and merge them together, two at a time, until one final combined history remains. The *Merge* algorithm is outlined in algorithm 1.

Algorithm 1 The ***Merge*** algorithm

```
function MERGE(history1, history2) returns a merged history
    combinedHistory ← UNIONNODESANDEDGES(history1, history2)
    for all action in combinedHistory do
        if ISOUTGOING(action) then
            for all action' in combinedHistory do
                if MATCHES(action, action') then
                    combinedHistory ← ADDEDGE(action, action', combinedHistory)
                end if
            end for
        end if
    end for
    return combinedHistory
end function
```

Figure 4.3 shows the local histories of two events, before and after a matching pair of actions have been found.

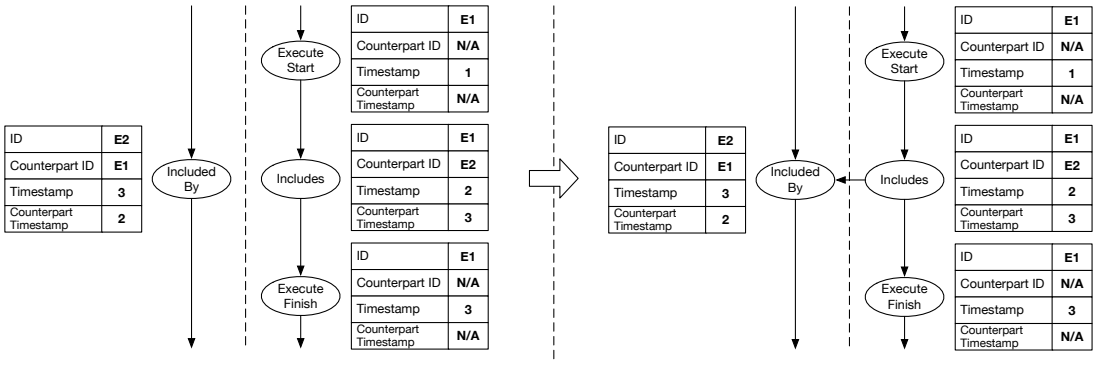


Figure 4.3: Actions of two merged histories before and after being matched.

In order to show that we have solved the problem of merging histories, we must show that all existing happens before relations of the local histories are still preserved and that we have added all of the possible happens before relations across histories.

Since the method takes the union of all of the histories, which implies the union of all the action sets and happens before sets, and because no actions or happens before relations are removed, all the existing actions and happens before relations are preserved in the global history, which means that all paths in the local histories are present in the global history.

Given that the union of all local histories contains the actions of every message exchange that has happened in the workflow and the algorithm have found all matching actions, corresponding to every message exchange, then every message exchange must be part of the global history. Because it is only possible to find happens-before relations across processes when they have exchanged a message, all happens-before relations between the histories of events must have been found.

To explain why it is necessary to assume that all local histories are valid, we must first describe the implications of local histories being invalid. If we do not assume that all local histories are

valid, it is not possible to assert that the union of the local histories contains all the actions and happens before relations between the actions that have happened, and therefore the global history will not be a representation of the actual global history of the workflow. This also means that we cannot assert that message exchanges, and actions in general, are not created, omitted, or changed. In the end, the algorithm could return an invalid global history if we do not assume that all local histories are valid.

4.2 Gathering of Local Histories

To merge the histories together it is necessary to acquire the local histories of the events of the DCR graph.

4.2.1 Gathering With a Central Server

In a central server system where the server has access to all the individual events, it is a trivial task to gather the histories. Simply request the server to get access to all the events and then request the histories of each event individually. This method is simple and only relies on the central server to get the addresses of all the events in the workflow. Furthermore the amount of messages sent to recieve all histories is $2N + 2$ where N denotes the amount of events in the system. This is because the client has to send one message to the server and receive a response from it followed by a message exchange with each of the events. An illustration of this architecture is shown on figure 4.4.

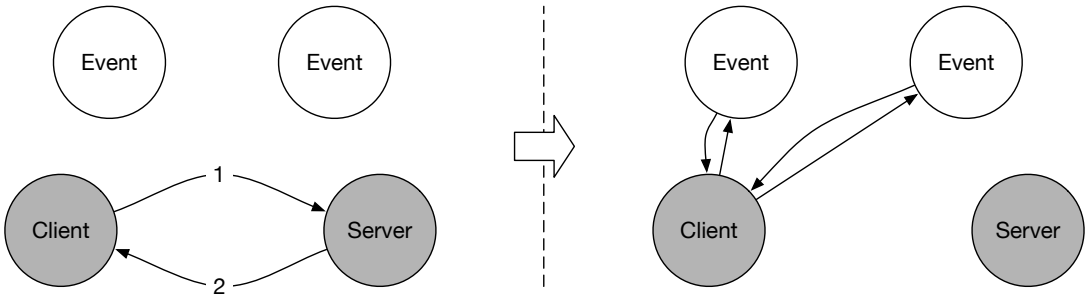


Figure 4.4: The client requesting history from events and receiving it.

Unfortunately, central server systems have drawbacks. E.g. if the server crashes or if it is busy processing other requests, then it is not possible to acquire the addresses of any of the events.

Therefore it is desirable to create a peer to peer-based gathering algorithm, but this introduces a new set of challenges.

4.2.2 Gathering Without a Central Server

In order to use a peer-to-peer algorithm to gather history from a given workflow, it is necessary to study the structure of the distributed DCR graph.

In distributed DCR graphs, events are connected only by their relations. Since there is no overall policy governing the placement of events on nodes in the network, a distributed DCR

graph is unstructured. Therefore the only way of contacting events, is using recursive traversal of relations of events, and a single starting event must be known. The reachability of this event should preferably be the set of all other events in the workflow. If the reachability is a subset of all the events, then only a subset of all the local histories of the events can be gathered. Therefore the greater the reachability of the beginning event, the more complete the gathered history will be. This implies that it cannot not be guaranteed that the history of every event will be gathered if the DCR graph is not fully connected. This constitutes the first drawback of this approach.

It is desired to develop an algorithm that, given an event with the desired reachability, is able to recursively gather all the local histories of the workflow. Since DCR graphs can contain cycles the algorithm also has to avoid endless recursion.

Given a DCR graph where all events are reachable from the starting event, the algorithm should gather and merge the histories of each event in the graph into one global history.

This is accomplished by recursively requesting the histories of neighbouring events and afterwards merging these histories with its own. To handle cycles, the use of a *request trace* of previously contacted events is sent from each event to the next.

When an event receives a request for its history, it subtracts all the events of the `request trace` from the set of the neighbours of the event. The result is the set of events to request history from. If this set is empty, the history of the event itself is returned. If the set is not empty, the event sends requests to all of the events with its own ID appended to the `request trace`. When the event receives histories from its neighbours, it merges the histories and returns the merged history to the requester. The algorithm can be seen in algorithm 2 and a run of the algorithm is illustrated in figure 4.5.

Algorithm 2 The ***Produce*** algorithm

Event receives request for history with a `request trace`, T.

```
function PRODUCE(T, event)
    waitFor ← MINUS(event.Neighbours, T)          ▷ Subtract T from neighbours.
    if waitFor = ∅ then return event.History
    else
        T' ←ADD(event.Id, T)

        gatheredHistories ← event.History
        for all event' in waitFor do
            history' ← PRODUCE(T', event')        ▷ Requests history of event' recursively
            gatheredHistories ← MERGE(history', gatheredHistories)
        end for
        return gatheredHistories
    end if
end function
```

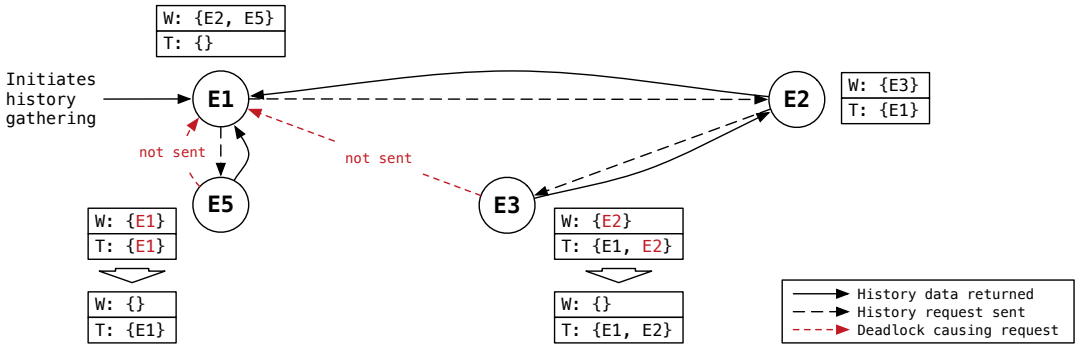


Figure 4.5: An illustration showing the recursive traversal the DCR graph seen on figure 4.6. Note the avoidance of creating a deadlock.

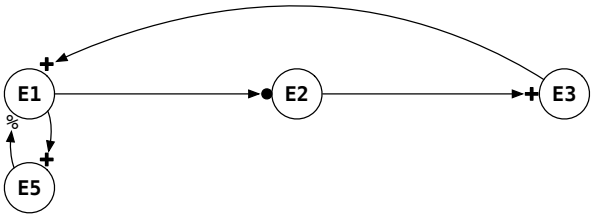


Figure 4.6: An example of a DCR graph with cycles.

This algorithm is in fact a modified version of the *flooding search algorithm*, described in [4], for searching in unstructured peer-to-peer networks. Contrary to flooding, the produce algorithm not only returns the value at the searched for node, but returns every value at every reachable node. The flooding algorithm is often implemented with a time limit on the search and therefore does not care about cycles, which is handled more efficiently in the produce algorithm. Furthermore, in the produce algorithm, every path to a node returns the value of that node.

To argue that the produce algorithm correctly gathers all the local histories in the workflow, we must show that every event is contacted. Since the algorithm contacts the neighbour of every contacted event, and the beginning event has a reachability of every event in the workflow, the set of contacted events must be all events in the workflow. If it is assumed that every event returns their history, as well as any recursively contacted event, then every history of the workflow must be returned to the beginning event.

The assumption that every event returns the history of every recursively contacted event is unsafe when malicious processes exist in systems, which constitutes the second drawback of the produce algorithm.

Issues with the Produce Algorithm

If malicious processes are introduced in the DCR graph, a problem similar to not having full reachability from the beginning event arises. If the path to a given event goes through an event on a malicious process, then that malicious process can either add, remove or change the history.

That implies that any history retrieved from that event might not be valid. The effects of this malicious behavior is worsened if the path to a great number of events goes through a malicious process. An illustration of this effect is shown in figure 4.7. If there exists no path to an event that does not go through a malicious process, then the beginning node will potentially not have any valid version of that history. Even if it is possible to assert that the history is invalid, it is not possible to identify which event along the path has acted maliciously.

Furthermore, this uncertainty is amplified, since for each contacted event with lacking information it is possible to tamper with the history of the recursively called events, or even add history of non existing events. Therefore some action must be taken to handle these pitfalls.

One way to handle processes changing and adding histories of other events, is to sign the histories before returning them to the requester. This is the approach for solving the Byzantine Generals Problem. Since the process starting the produce algorithm does not know the address of all events, then it is possible for other processes to sign changed or added histories with its own certificate and tell anyone who want to confirm the certificate to contact the malicious process itself. In order for these certificates to be effective, all processes must therefore know the certificate or location of any other process in the system.

If the certificates of all processes are known beforehand, a security breach would leave the entire system open for attacks. On the other hand, if the addresses of all events are known, then we have a situation where the central server algorithm is sufficient since certificates of messages are no longer required, under the assumption that man-in-the-middle attacks does not occur.

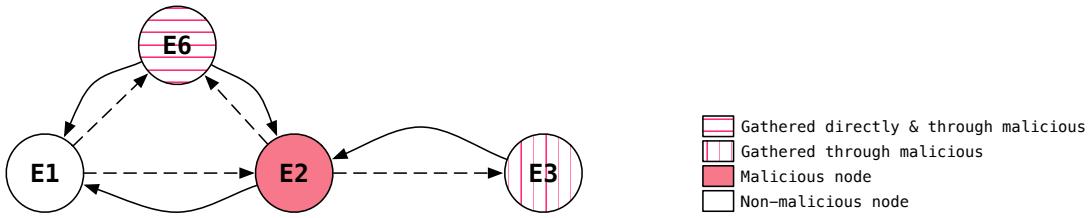


Figure 4.7: A malicious node in a graph. Note that it is not possible to ensure that any history passing through the malicious node is valid.

The produce algorithm sends an exponential number of messages in the amount of events in the worst case. Redundant history from every path from a given node to any other reachable node is wanted. This is the reason that the complexity is exponential. The worst case is when the graph is fully connected and every node therefore has an edge to any other node, implying that every permutation of nodes are created by the algorithm. High connectivity in the graph is a wanted property, since it allows for better validation of the histories. This is explained in chapter 6.

Caching history produced of an event before transferring it to a requesting event could improve performance, but presents a new problem. This problem is most easily illustrated with a figure. In figure 4.8 we have a situation where four events, A, B, C, and D gather histories. The numbers on the arrows in the figure represent the order in which messages are sent. A starts by contacting B, which in turn contacts C which contacts D. D returns its local history to C because B is already part of the trace. C then caches the result of D and its local history. C then returns the history to B, which caches the result along with its local history. B returns the

history to A, which can then contact C for its history. When C is contacted, it just returns its cached history, which does not contain the history of B. The history of B can therefore not be confirmed on multiple paths.

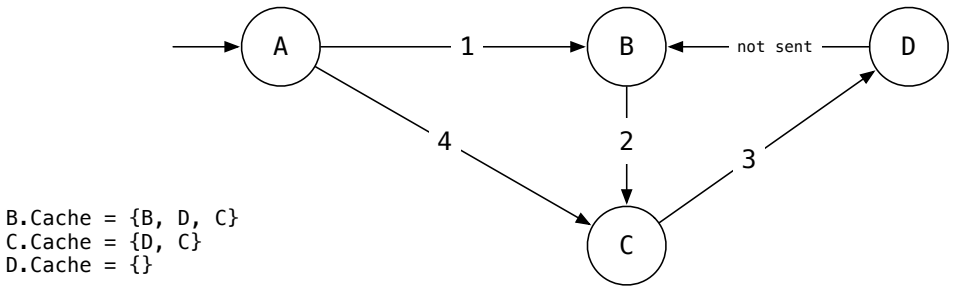


Figure 4.8: An example of a recursive gathering where caching would result in a incomplete history being returned.

There might be solutions to the presented issues, but these have not been examined further in this project. Because of the problems presented in this section, we have chosen to use the central server gathering algorithm.

Chapter 5

Reducing Histories to Orders of Execution

In this chapter it is examined how it is possible to find an order of execution from a valid global history and it is shown that consensus can be reached on the found order of execution with an election.

5.1 Order of Execution

A valid global history represents every action that has occurred on all the events of the workflow. It is desired to find an order of execution with excess information in the history removed without losing information regarding the order of the executions. To do this, we need to represent an execution as a single entity and furthermore find the minimum equivalent graph of the resulting order.

5.1.1 Single Entity Representing an Execution

In order to find a single representation of an execution, one approach could be to examine if a single action of an execution could describe all the relations to all the other executions. Doing this involves filtering all actions not chosen to be the representative of the execution, and since filtering nodes also removes their edges, happens-before relations between executions would be lost. Therefore it is necessary to add a direct edge between representatives if there is already a path from one to the other in the history. Since the history is represented as a directed acyclic graph, the transitive closure of the graph provides the desired relations. Having found the transitive closure of the history, actions that are not the representative for an execution are removed from the graph along with their edges.

Since every execution has actions of types *Execute start* and *Execute finish*, choosing one of these as the representative seems like a sensible choice. For any execution in a history, an action of type *Execute start* always has a path to an action of type *Execute finish*. Therefore, the transitive property of directed acyclic graphs states that in an execution the reachability of an action of type *Execute start* is a superset of the reachability of the action of type *Execute finish*, and therefore it contains more information about happens before relations. To conclude, actions of type *Execute start* are the most sensible choice of these two types.

We now extend the legend from figure 3.1 with the following illustration types:

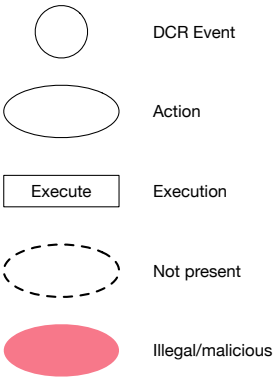


Figure 5.1: A legend showing what the different figures mean in the following sections.

A situation where actions of type *Execute start* are chosen as representatives is shown in figure 5.2. What becomes apparent in this figure, is that the first execution of event A must happen before the first execution of event B. This is because two events that have a happens-before relation must have any kind of relation in the workflow. If the events have a relation in the workflow, then they must execute serially equivalent. This ensures that an execution of one of the events must happen either before, or after an execution of the other. Therefore another approach must be taken.

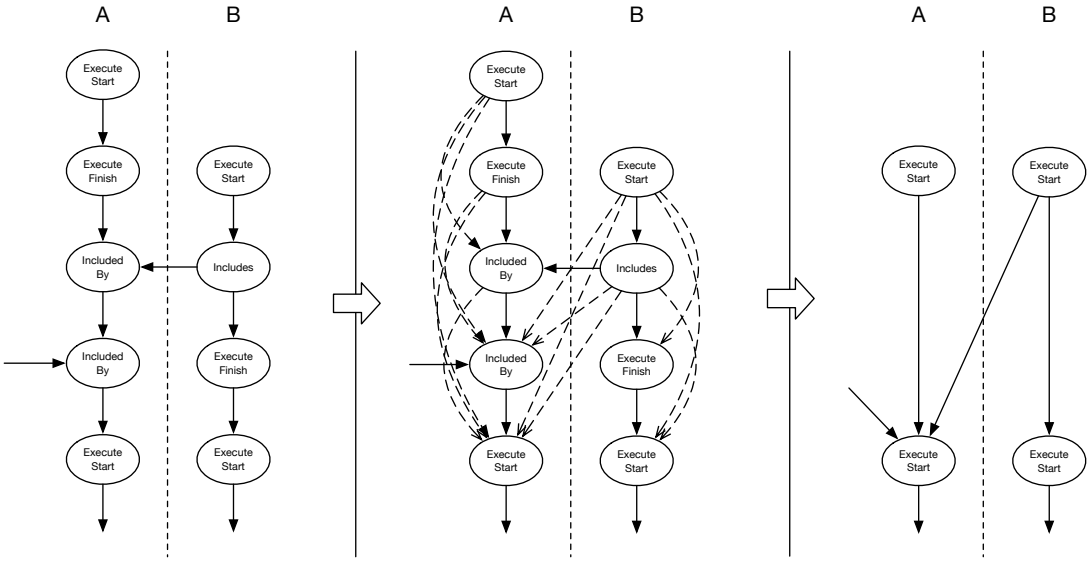


Figure 5.2: The result of finding the transitive closure on from the actions of an event. Note that the topmost executions happen concurrently. So do the two lower executions.

In [13], Sedgewick et. al. describes an implementation of transitive closure, which has a time complexity of $\mathcal{O}(V * (V + E))$ where V is the number of nodes and E is the number of edges in the graph.

5.1.2 Collapsing

As described above, transitive closures are not an ideal way of representing an execution as a single entity and therefore collapsing is introduced.

In definition 5 an execution is defined to be all the actions representing the effects of the execution, as well as two actions with types *Execute start* and *Execute finish*. Even though incoming actions are stored on other events than the ones performing the executions, these actions are still initiated by the executing event and are therefore seen as being part of the execution. Since an event is only affected by the effects of other events or itself executing, any action is always part of *some* execution of *some* event, implying that no action can happen outside an execution.

In a global history, given an action of type *Execute start*, the set of actions representing a single execution is found by taking each consecutive action with the same event ID until an action of type *Execute finish* is found. Furthermore, the incoming actions corresponding to the outgoing actions already found in the execution are added to the set.

In invariant 1 it is said that if a single action in one execution happens before another action in another execution, then no action from the second execution can happen before an action in the first. In other words, this means that if there exists an action in execution A that happens before an action in execution B, then A happens before B and B cannot happen before A. Note that this does not mean that all actions in A happen before all actions in B, which means some of the actions are in fact concurrent. This creates the need for an argument that A *does* happen before B, despite that some actions are concurrent.

As stated in the previous section, because of serial equivalence, two events, where one has a relation to the other, can only execute in a way such that the result is the same as if one had executed before the other. Therefore any happens-before relation any action in an execution has with any action outside of that execution actually applies to all actions in the execution. When this is the case, every action inside an execution is equal in terms of happens-before relations and therefore we can *collapse* the execution into one single entity.

An example of collapsing actions into executions is shown in figure 5.3. This is the same example as used when describing transitive closure. Contrary to figure 5.2, the first execution of A happens before the first execution of B.

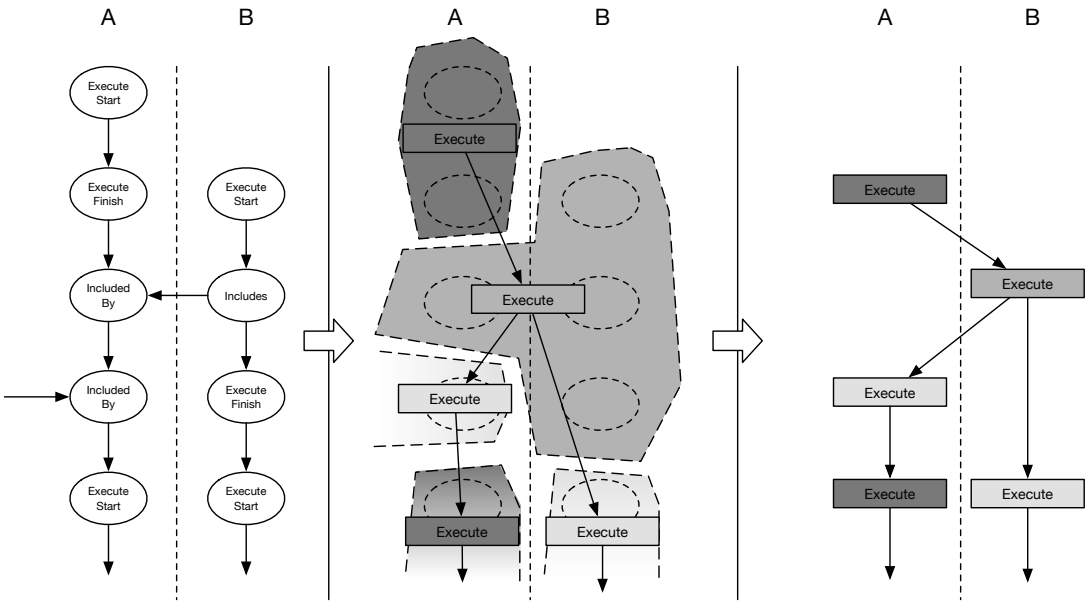


Figure 5.3: The result of finding an execution using *Collapse*. Note that some executions that previously were concurrent (see figure 5.2) are now ordered.

The collapsing algorithm can be be seen in algorithm 3. The algorithm gathers all actions of an execution and, after the single entity execution is created, creates all happens-before relations. In order to do so, it uses a few helper functions. `GetEdgesToOtherExecutions` together with `MapOldEdgesToNewIDs` represent the translation of edges of actions in one execution to actions in another execution, to edges from execution entity to execution entity.

Algorithm 3 Collapse algorithm

```
function COLLAPSE(history) returns an order of execution
    actionMapper ← EMPTYMAP( )
    executeStartActions ← FILTERBYTYPE(ExecuteStart, history)

    for all action in executeStartActions do                          ▷ Loop 1
        execution ← FINDSINGLEEXECUTION(action)
        uniqueId ← GENERATEUNIQUEID( )
        for all executionAction in execution do
            actionMapper ← ADD(action.Id, uniqueId, actionMapper)
        end for
    end for

    orderOfExecution ← EMPTYGRAPH( )

    for all oldActionID, newActionID in actionMapper do               ▷ Loop 2
        oldEdges ← GETEDGESTOOTHEREXECUTIONS(oldActionId, history)
        edges ← MAPOLDEDGESTONEWIDS(oldEdges, actionMapper)
        newAction ← CREATEACTION(newActionId, edges)
        orderOfExecution ← ADDNODE(newAction, orderOfExecution)
                ▷ This requires ADDNODE to merge edge sets when existing nodes are added
    end for
    return orderOfExecution
end function
```

One of the properties the collapsed order of execution has, is that, for two executions to be concurrent the executing events must be independent, as described in [5], however, not all independent events create concurrent executions.

All non-independent events cannot execute concurrently, and to demonstrate this, we will show that if any of the properties of independent events are not present between a pair of events, therefore making them non-independent, then collapse will order the executions.

1. *No event included by E is excluded by F and vice versa*: This property is violated if E includes an event, G that F excludes. The incoming actions on G will allow collapse to order the executions of E and F.

2. *E requires a response from some G iff F does*: This property is violated if E has a response relation to F, but F does not have a response relation to itself. In this case, an execution of E results in incoming actions on F. Any execution of F is always before or after these incoming actions.

3. *Neither event is a condition for the other*: This property is violated if E is a condition for F. In this case the incoming actions from the execution of F on E, enables the ordering of executions of the events.

4. *Neither event includes or excludes the other*: This property is violated if E includes F. In this case the incoming actions from the execution of E on F, enables the ordering of executions of the events.

5. *Neither event includes or excludes a condition of the other*: This property is violated if E excludes G which is a condition of F. Recall that conditions in the implementation are

checked by the executing event. This means that any execution of F will result in incoming actions on G. Also the execution of E will result in incoming actions on G. Therefore the incoming actions on G will allow collapse to order the executions of E and F.

To argue that the collapse algorithm correctly describes the relations between executions invariant 1 is examined. The invariant states that if there exist any action in execution A with a relation to execution B, then it must not be true the other way around. Therefore for collapse to be correct every single execution entity must have the abide to the invariant.

Since the set of relations the entity representing an execution has to other executions is the union of the sets of all the relations the actions of the execution had to other executions, then the set cannot contain relations which violates the invariant, because that would imply that the invariant would already be violated before collapsing. If the invariant is violated in the union set, then there must have been an action in an execution A that happens before an action in an execution B and an action in B that happens before an action in A in the original collection of sets representing the executions.

The algorithm runs in a time complexity of $\mathcal{O}(N + E)$ where N is the number of actions and E is the number of edges in the history. This is because `FindSingleExecution` and `executeStartActions` are inversely proportional and together constitutes the amount of actions in the graph. Therefore the overall complexity of `loop 1` is linear to the amount of actions. `Loop 2` iterates over every action in the history, furthermore every edge in the history is iterated over since they are retrieved for every action with `GetEdgesToOTherExecutions`. `MapOldEdgesToNewIds`, `CreateAction` runs in constant time and `AddNode` runs linear time to the amount of edges between executions, then the first part of the second loop is the dominant factor. Therefore the second loop run in $\mathcal{O}(N + E)$, and is then the most dominant factor of the entire algorithm.

5.1.3 Transitive Reduction

Redundant edges might still exist in between executions after collapsing a history. An edge is redundant in cases where there exists a non-direct path from an execution to another execution, but there also exists a direct edge between the two executions. In this case the direct edge is redundant and can be removed, since it does not contribute extra information in regards to the ordering of the executions, as it only explicitly expresses what is implicitly available. Removing the direct edge will not affect the ordering or reachability of the order of execution, but rather simplify it.

Figure 5.4 illustrates a case where a redundant edge exists between executions of event B and A, since there is a path from the execution of event B to the execution of A through the execution of event C. Recall that the order of execution is represented as a directed acyclic graph and that it is possible to find a minimal equivalent graph for such a graph.

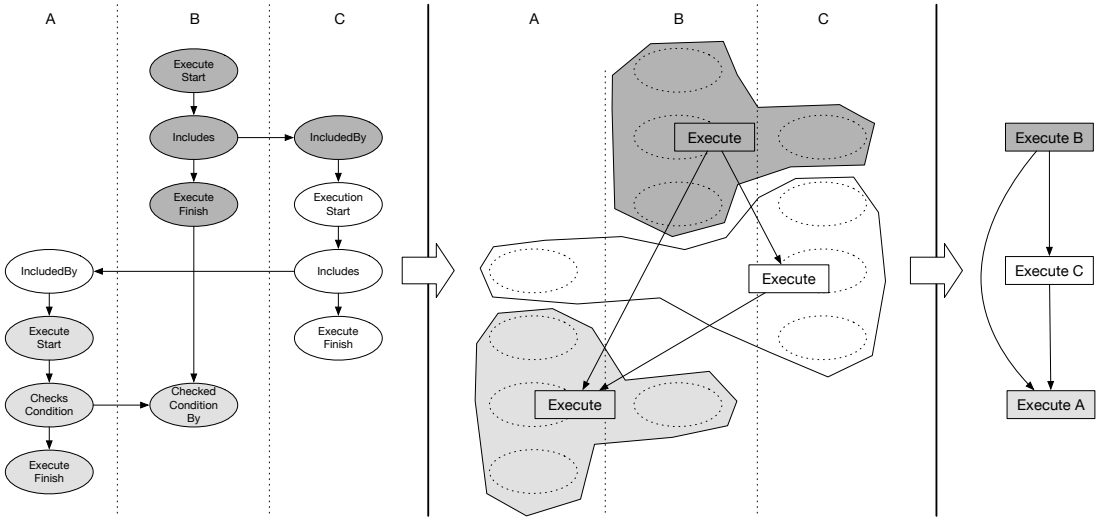


Figure 5.4: The result of collapsing actions in the histories of three events.

In directed acyclic graphs the transitive reduction produces the minimal equivalent graph. A naive algorithm for finding the transitive reduction on an order of execution is shown in algorithm 4.

Algorithm 4 Transitive Reduction Algorithm

```
function TRANSITIVE-REDUCTION(orderOfExecution)
    for all execution1 in orderOfExecution do                    ▷ Loop 1
        for all execution2 in orderOfExecution do                ▷ Loop 2
            if PATHEXISTS(execution1, execution2, orderOfExecution) then
                for all execution3 in execution2.edges do        ▷ Loop 3
                    if EDGEEXISTS(execution1, execution3) then
                        orderOfExecution ←
                            REMOVEEDGE(execution1, execution3, orderOfExecution)
                    end if
                end for
            end if
        end for
    end for
    return orderOfExecution
end function
```

The shown algorithm for transitive reduction has a time complexity of $\mathcal{O}(n^2(n+e))$ where n is the number of executions and e is the number of happens-before relations between executions. Since `EdgeExists` and `RemoveEdge` run in constant time, `loop 3` has a time complexity of $\mathcal{O}(E)$ because in worst case `loop 2` iterates over all executions in the graph and then `execution2.edges` represents all edges in the graph. `PathExists` has a time complexity of $\mathcal{O}(N + E)$ because each node is visited over each edge. Therefore `PathExists` becomes the dominant factor of the

second loop. `Loop 1` and `Loop 2` has a time complexity of $\mathcal{O}(N^2)$ and therefore the combined time complexity of the entire algorithm is $\mathcal{O}(n^2(n+e))$.

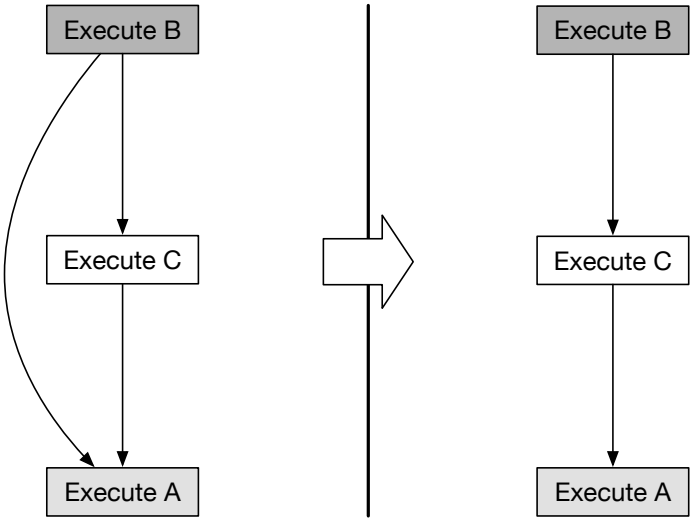


Figure 5.5: The result of reducing an order of execution with redundant edges.

As with the input to the merge algorithm presented in chapter 4, the input of the collapse algorithm must be a valid global history for the algorithm to produce a valid order of execution. If the global history is not valid, then the validity of the order of execution cannot be guaranteed. As the transitive reduction algorithm only remove redundant information from an order of execution, the same problem arises.

We will see in chapter 6 that an order of execution can be used to validate certain kinds of cheating.

5.2 Election

Assuming that the found order of execution is valid, we want to know if the processes hosting the events of the workflow can reach consensus on this order of execution.

Recall the three requirements for a consensus algorithm: Termination, agreement, and integrity. We want to make an election that fulfils these requirements.

For this consensus problem, the proposed value is the global order of execution and for a process hosting an event to accept a proposed global order of execution, it must be able to map all the actions of its local history onto the executions in the global order of execution.

To map actions in the local history to an order of execution the following procedure is followed:

- If an action has the type *Execute start*, then all consecutive actions until an action of type *Execute finish* is found, are part of the same execution of the event itself.
- If an incoming action is found, then all following actions with the same counterpart ID but distinct action types represent an execution of the counterpart event.

– This implies that if a new action with the same counterpart ID and an action type not distinct from the ones before it, the action must represent the beginning of a new execution of the counterpart event.
– It also implies that if an incoming action with another counterpart ID is found, then this is the beginning of an execution of the event with the new counterpart ID.

Because the mapping is created consecutively and because the local history is in total order, the mapped executions must have the same happens before relations as the order in which they are found. Furthermore, if one execution of an event exists in the mapping of the local history, then all executions of that event must be represented in the local history. Therefore the first mapping of an execution in the local history must be the execution of the specified event that happens before any other execution of that event in the global history.

An example of mapping actions of local histories to executions are shown in figure 5.6. In the figure it is seen that pairs of actions with types *Execute start* and *Execute finish* represent executions of the local event. Furthermore, because event A included event B twice, two executions of A must have happened. Also note that the order of the actions are preserved in the local order of execution.

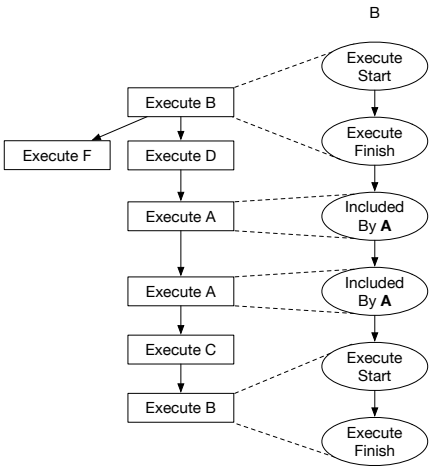


Figure 5.6: An example of mapping actions of local histories to executions

From any mapping of an execution found in the global order of execution it should be possible to find the next mapping of an execution only by traversing edges in the global order of execution.

If the local history matches the global order of execution, then the process hosting the event accepts the proposed value.

We have now described how a proposed value can be voted on. It still needs to be discussed how an election is initiated.

As described in chapter 4, the client retrieves all local histories of a workflow and then it merges, collapses, and reduces the histories into an global order of execution. Therefore the client is the only process able to propose a value that all processes hosting events can reach consensus on. Because the client already knows all events in order to retrieve their histories, it can propose the global order of execution to each of them and await their answers.

If the local histories are valid and all of the presented algorithms are correct, then all processes hosting events will reach consensus on the proposed global order of execution, since the local orders of execution are preserved in the global. Therefore both the agreement and integrity requirement of the election is fulfilled. Because we do not allow processes to crash while carrying out a task, every process must at some point respond to the proposal thereby setting its decided value. Therefore the termination requirement is fulfilled.

If any process does not vote in favour of the order of execution, it implies that either a local history was invalid, the proposer created an incorrect order of execution, or that the process incorrectly voted against the proposed value. In either of the cases, a malicious process must exist in the system, and therefore if no process votes against the proposed value, we have solved the problem of reaching consensus on an order of execution of an execution of a distributed DCR graph when no malicious processes are present in the system.

Chapter 6

Validation of Histories

This chapter defines how histories from events on malicious processes are observed and possibly identified.

We introduce malicious processes in distributed DCR graphs. The implementation allows multiple events to be hosted by a single process, but in this chapter it is assumed that any process only hosts a single event. Malicious processes corrupts data in many ways and therefore it is desired to have a mechanism which can observe and, if possible, identify these corruptions and the processes which created them in a predictable way.

As mentioned in definition 7 the valid history has the following properties:

1. Actions in the history are not invented, changed or removed.
2. The rules of the workflow are followed.
3. The history abides to serial equivalence.
4. The history is in a strict partial order.

Cheating is the act of creating histories which violates at least one of these properties. This leads us to two different kinds of cheating:

Definition 10. ***Inconsistent cheating*** *violates at least one of properties 2, 3, and 4.*

Definition 11. ***Consistent cheating*** *violates property 1, but not 2, 3, and 4.*

Definition 10 defines that inconsistent cheating is the act of creating a history which could not have happened according to the rules of the workflow, serial equivalence or strict partial ordering of actions. Definition 11 defines that consistent cheating is the act of creating a history which could have happened, but did not.

A malicious process is a process which participates in consistent or inconsistent cheating.

6.1 Inconsistent Cheating

To examine inconsistent cheating in depth, we must explore cheating where each of the properties 2, 3, and 4 are violated. We will see that either property violation can be identified separately.

6.1.1 DCR Rules

A given local history must abide to the rules of the workflow it has been created from. In order to validate a local history against these rules, the events, their relations and the initial state of the workflow must be known. The following sections each describe an assertion that can be made in order to guarantee that the history has followed the rules of the workflow.

As seen in chapter 3, actions are based on the effects that happen when an event executes. Actions match the relations of a workflow if the event ID, counterpart ID and action type correspond to the start node, end node, and type of the relation in the DCR graph. We will use this notion to explain the DCR rules that a history must follow.

Rule 1. ***Only specified relations****: The local history of an event must only contain actions of corresponding relations in the workflow.*

An example of a DCR graph and the local histories of the events are shown in figure 6.1. All events, A, B, and C, contain relations that are not specified in the workflow.

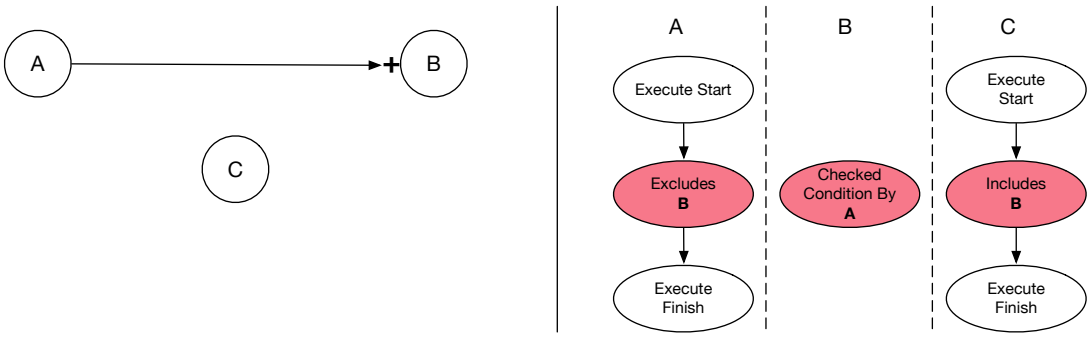


Figure 6.1: A DCR graph and three examples of local histories violating rule 1.

It can be validated that local histories adhere to rule 1 by examining every action, and for each outgoing and incoming action, the event, its counterpart, and action type must be present in the rules of the workflow. If an invalid relation is present then either the event or the counterpart of the relation must be hosted on malicious processes. Therefore this kind of cheating is observable but not identifiable.

Rule 2. ***Only complete executions****: A given event, E, must have actions for each of its outgoing relations when executing. Each of the counterparts of these actions must have a corresponding action in their history after E has executed. For any execution there must be exactly one action of type* Execute start *happening before a single action of type* Execute finish*.*

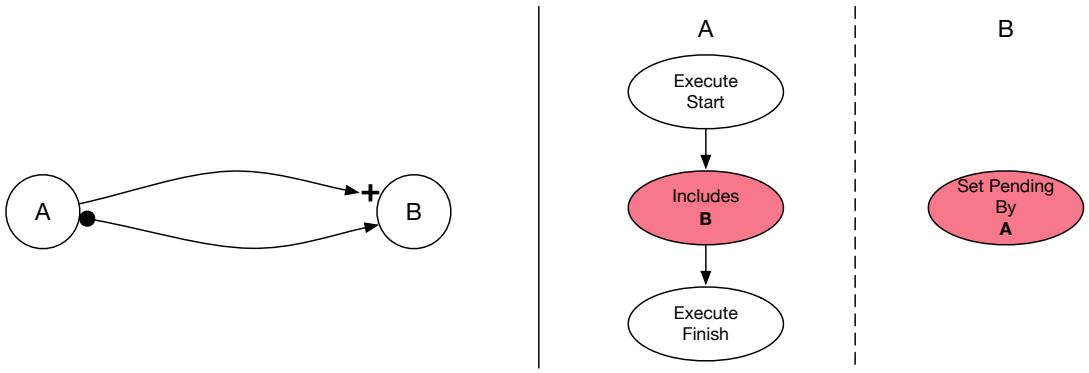


Figure 6.2: A DCR graph and two examples of local histories violating rule 2.

In figure 6.2 event A does not contain an outgoing action of type *Sets pending* and event B does not contain an incoming action of type *Included by*.

In order to validate rule 2 for outgoing relations the following approach is taken: When an action of type *Execute start* is encountered in the local history, a set of required actions for the execution is created from the rules of the workflow.

For example, if event A includes event B and excludes itself, this set must contain an outgoing action corresponding to the inclusion of B, and two actions, one outgoing and one incoming, corresponding to the exclusion of event A itself.

Each consecutive action in the history, must be one of the actions in the set. The action is then removed from the set. When the set is empty, the next action in the history must have type *Execute finish*.

In order to validate rule 2 for incoming relations where the counterpart is not the event iself the following approach is taken: When an incoming action from an event A is found, all actions corresponding to the remaining rules that specify that A has a relation to the current event is added. While the set is not empty the consecutive actions in the history must exist in the set. An action is removed from the set when it is encountered.

Violation of this rule is observable but not identifiable since an event does not know what relations other events has to it. Therefore a malicious process could only effects some of its relations but add actions as if it had done it, making it impossible to identify which of the processes are malicious.

Rule 3. ***Executions only in Valid States****: The history of an event must only contain executions when the event is executable. This implies that the entirety of the history must match a run of the DCR graph it represents.*

Checking for this kind of inconsistency is nontrivial. A method of doing so is described in section 6.3.

To argue that rules 1 through 3 covers the DCR rules, the definition of DCR graphs is examined. In section 2.1 the definition of a DCR graph is described. From the definition it is possible to make a set of all relations where each element is a triple of two events and the type of the relation. Because this set is finite, rule 1 makes sure that the actions can only represent the relations that

exists in that set. Furthermore, since the relations can be filtered by events it is possible to get the set of relations that happen in an execution. This is ensured by rule 2 which makes sure that every action that should happen, happens in an execution. The definition of DCR graphs also declares what the state of the workflow must be for an event to be executable and we will see in section 6.3 that this case is also covered. The definition of a DCR graph does not have any more requirements for the run of a workflow and therefore these rules must cover all kinds of cheating which contradicts the rules of the workflow.

6.1.2 Serial Equivalence Rules

A given execution must comply to the rules of serial equivalence. That includes the following:

Rule 4. ***Only non-disrupted executions****: A given event, E must not contain ingoing relations or execute starts when already executing. Furthermore if an execution is started, it must also finish. The exception to the rule of ingoing relations is when an event has a relation to itself in which case that incoming action is allowed.*

Figure 6.3 shows an example where an execution of event A is interrupted by an incoming action from another event.

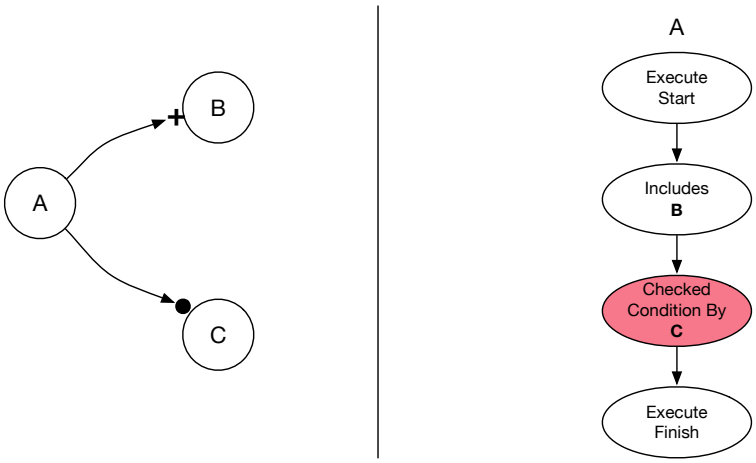


Figure 6.3: A DCR graph and an example of a history where rule 4 is violated

The algorithm used to check rule 2 for outgoing relations is also used to check rule 4.

Rule 5. ***Wait for complete execution****: A given event, A must be affected by all its ingoing relations from an event, B when B executes before anything else happens. The exception to the rule is when an event has a relation to itself in which case actions from the execution are allowed before the next incoming action happens.*

Figure 6.4 shows a situation where B is included by A but executes before the response relation from A is present in the history.

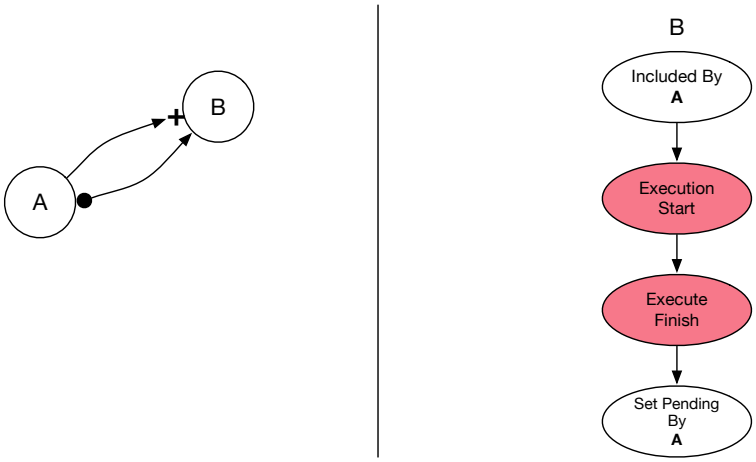


Figure 6.4: A DCR graph and an example of a history where rule 5 is violated

The algorithm used to check rule 2 for incoming relations is also used to check rule 4.

Rule 6. ***Synchronous Message Exchanges****: If message exchanges happen synchronously outgoing or incoming actions must have counterpart timestamps higher than any counterpart timestamp of actions that happen before it.*

A validation of this kind of inconsistency can be done on each single event. Since each incoming and outgoing action must have a counterpart, it is possible to keep a mapping between counterparts and the last timestamp that counterpart had. Recall that the implementation requires that all correct processes only accept higher timestamps for a counterpart, than any timestamp seen from that counterpart before. If a new timestamp for a counterpart is lower than the previous, then the event must be malicious.

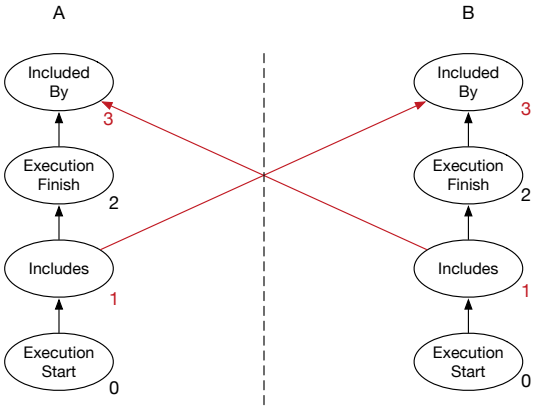


Figure 6.5: An example of two local histories violating rule 6.

Figure 6.5 shows a situation where the counterpart timestamps are not in order on neither event A nor B which violates rule 6. Because the implementation requires serial equivalent

executions, and because message transfers happen synchronously, that is the sending process receives confirmation that the message has been received before continuing the execution, such a violation will never happen on a correct process, as this would imply a pair of executions which happens before each other. Imagine the situation on the figure 6.5, event A includes B and sets the counterpart timestamp to 3. Then it finishes its execution and gets included by B, but it receives a counterpart timestamp of 1. This must imply one of two things, either the timestamps on event B are out of order, or event B has been executing and accepting incoming actions simultaneously, creating a situation which is not serially equivalent.

To argue that rules 4 through 6 cover the requirements of serial equivalence, recall that serial equivalence requires that any interleaving of transactions must create the same result, as if the transactions had happened in sequence. The implementation of a correct process in distributed DCR graphs must therefore only allow executions of events in a serially equivalent manner. As described in section 2.4, the provided implementation uses locking to achieve serial equivalence and message exchanges happen synchronously. In the implementation an action of type *Execute start* is saved when a process has received all locks on events it affects during the execution of an event. Also, an action of type *Execution finish* is saved just before all locks are released. This means, for all correct processes, any group of actions representing an execution must be performed when all affected events are locked. Furthermore, no actions from other executions must be present in between the actions of an execution. If this property is violated, this can be present in histories in the following three ways:

- An incoming action is inside an execution of another event. This is the case presented in figure 6.3 and can be found by applying rule 4.
- Any action, not part of an execution E, is between at least two incoming actions part of execution E. This is the case presented in figure 6.4 and can be found by applying rule 5.
- In a local history an action of type *Execute start* has a path to another action of type *Execute start* in which no action of type *Execute finish* exists. This case can be found by applying rule 4.

Together, these three ways of violating serial equivalence cover all cases of an event being affected while being locked. There is a case where two processes can participate in the creation of a history where each of the local histories are serially equivalent, but when the actions of the executions are connected, serial equivalence is violated because message exchanging was not done synchronously. Figure 6.6 illustrates a history and how it would look like if message confirmations were stored as part of synchronous message exchanges. This also explains why the situation in figure 6.5 is not valid, because a cycle would be present if message confirmations were explicit as seen in figure 6.7. Because of this, it can be concluded that both the execution of event A and B, happens before each other. Serial equivalence requires that the interleaving is equivalent to the execution of one of the events, followed by the other, it can be concluded that this interleaving is not serially equivalent. This last form of serially equivalence violation is covered by rule 6.

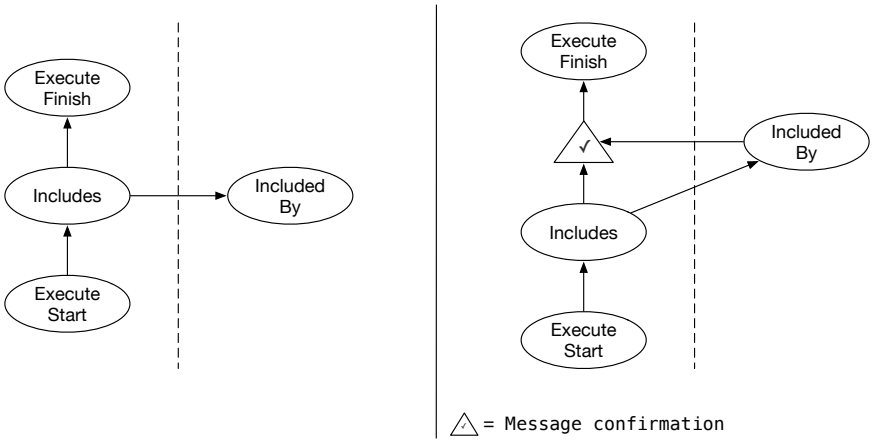


Figure 6.6: An illustration of how message exchanges correspond to synchronous communication.

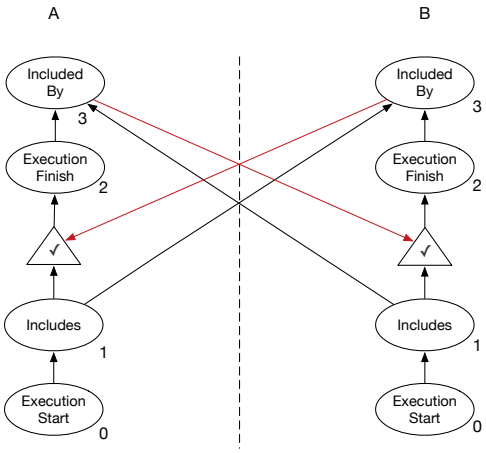


Figure 6.7: An illustration of cycles shown when message confirmations are made explicit.

6.1.3 Lamport's Logical Clocks

A given history must conform to the rules of Lamport's logical clocks. That includes the following:

Rule 7. ***Total Order of Timestamps****: Every action in the local history of an event must have a timestamp less than the timestamp of any action it happens before.*

This kind of inconsistency can be checked by examining each action on a single event. If the timestamps are not increasing for each edge in the graph then the process hosting the event must be cheating.

Rule 8. ***Message Exchange Timestamp Order****: For every pair of actions that represent a message exchange, the timestamp of the outgoing action must be less than the timestamp of the incoming action.*

A validation of this kind of inconsistency can be done on the history of each event. If the counterpart timestamps of the outgoing actions are not larger than the timestamps, then the process hosting the event is malicious. For incoming actions the counterpart timestamp must be less than the timestamp.

Because histories are represented as directed acyclic graphs, cycles must not exist. Rule 7 and 8 makes it possible to observe cycles and identify the local histories from where the cycles originate.

Together, these rules make sure that no cycles can exist. If an action happens before another action with a lower timestamp, at least one of rules 7 and 8 have been violated. Examples of violations of rules 7 and 8 are shown in figures 6.9 and 6.9.

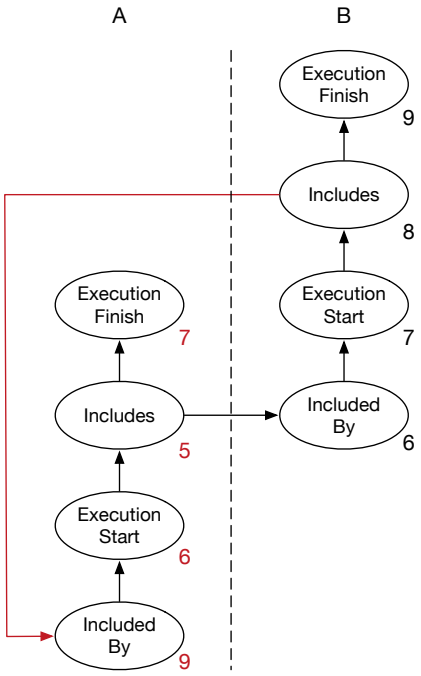


Figure 6.8: An example of two histories forming a cycle by violation of rule 7.

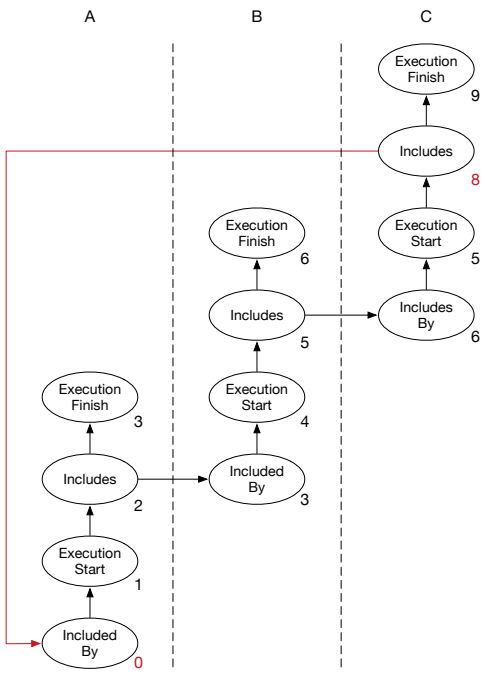


Figure 6.9: An example of three histories forming a cycle by violation of rule 8.

On figure 6.8 a cycle is created when event A changes the order of its actions, violating rule 7 while still keeping the timestamps of message exchanges in an increasing order.

On figure 6.9 a cycle is created when timestamps of message exchanges are not kept in an increasing order, violating rule 8. The timestamps of actions in the local histories are in increasing order which means that rule 7 is not violated.

To argue that rules 7 and 8 cover the requirements of Lamport's logical clocks, recall that Lamport states that the timestamp of a given action on a given process must be lower than any action it happens before. This is covered by rule 7.

Lamport also states that for any message exchange between two processes, the local timestamp of the process receiving the event, must be set to a higher timestamp than that of the

sending process. This is covered by rule 8. Together these two rules cover the requirements of Lamports logical clocks.

Inconsistent Cheating	**Observable**	**Identifiable**
Rule 1 (Only specified relations)	Yes	No
Rule 2 (Only complete executions)	Yes	No
Rule 3 (Executions only in valid states)	Yes	No
Rule 4 (Only non disrupted executions)	Yes	Yes
Rule 5 (Wait for complete execution)	Yes	Yes
Rule 6 (Synchronous message exchange)	Yes	Yes
Rule 7 (Total order of timestamps)	Yes	Yes
Rule 8 (Message exchange timestamp order)	Yes	Yes

Table 6.1: Table of the observability and identifiability of the different inconsistent cheating types

Table 6.1 summarises the observability and identifiability of violations of each of the rules 1 through 8. We will see in section 6.3 why rule 3 is observable but not identifiable. In the table it becomes apparent that in most cases it is possible to identify the single process that cheats, but in the cases where the violated rules regards the rules of the workflow, it is only possible to observe inconsistent cheating.

6.2 Consistent Cheating

Until now we have examined inconsistent cheating, but malicious processes are also able to cheat *consistently*. Recall that consistent cheating is the act of creating histories which *could* have happened but did not. In this section *original* actions and timestamps are actions and timestamps that did happen in the execution of the workflow.

Consistent cheating is present if one of the following rules are violated:

Rule 9. ***Agreement on original timestamps****: Two events must agree on the original timestamps of their actions corresponding to their relations to each other.*

When actions are matched together to form happens before relations between the histories of two events, the timestamps and counterpart timestamps are used to uniquely identify a pair of matching actions. If however it is not possible to match actions because timestamps are mismatched, then at least one of the histories must have been changed.

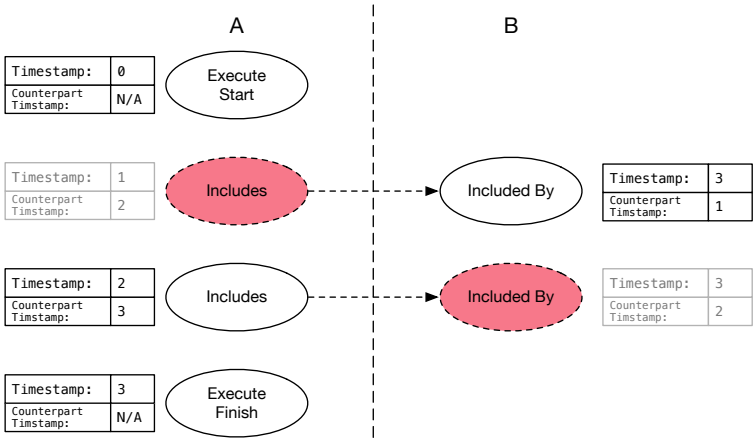


Figure 6.10: An example of two local histories with no inconsistent cheating, not agreeing on the timestamps thereby violating rule 9. Dashed actions marked with a red colour denote the actions which are not present but are expected by the counterpart.

An example of two events not having matching actions in their histories because of the timestamps can be seen on figure 6.10. Notice that it is not possible to distinguish the two include actions from each other, and that it is therefore not possible to identify which of the two histories are invalid.

To validate for this kind of cheating, pairs of local histories where a relation exists between them are examined. Each ingoing action is matched with outgoing actions, and if it is not possible to find a match, one of the processes hosting the events must be malicious. Since the original timestamps are unknown, it is not possible to distinguish an original timestamp from an unoriginal one, and therefore two events can agree on unoriginal timestamps without failing this validation.

Rule 10. ***Agreement on the original amount of executions****: If an event A has a relation to an event B, then the two events must agree on the number of executions of A in their histories.*

To check for this kind of cheating the local histories of each pair of events which are part of a relation are examined. If event A and event B are examined, then the amount of every outgoing action from A to B must match the amount of incoming actions on B from A. If the amount does not match then one of the processes hosting the events must be malicious.

On figure 6.11 an example of violation of rule 10 is shown. In the shown histories, event A states that it has executed two times but event B states that A have only executed one time. Similarly on figure 6.11 the two events do not agree on the amount of executions of A. But in that case Event A states that it only has executed one time, but B states that A has executed two times. Notice that it is not possible to distinguish the two situations from one another and it therefore is not possible to identify which of the two histories are invalid. Since the original amount of executions are unknown, two events can agree on an unoriginal amount of executions without failing this validation.

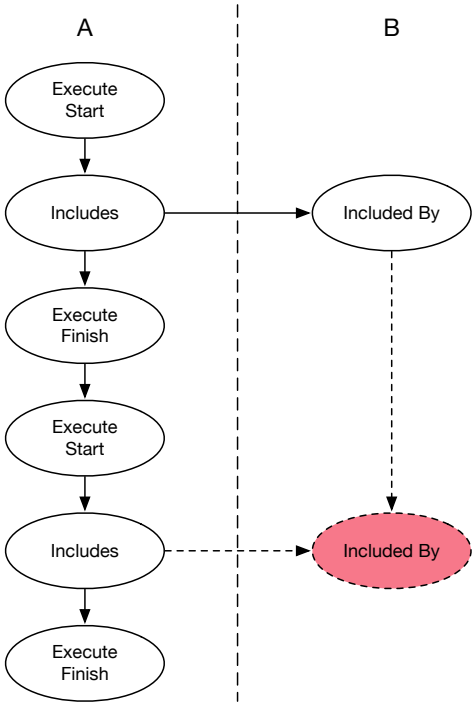


Figure 6.11: An example of two histories not agreeing on the amount of executions of A, violating rule 10.

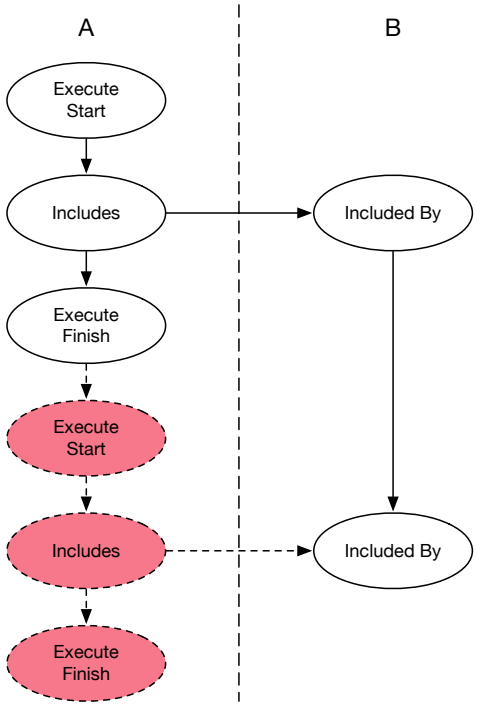


Figure 6.12: An example of two histories not agreeing on the amount of executions of A, violating rule 10.

Recall that the definition of consistent cheating specified that actions could be added, removed or changed but the resulting history must still have the properties 2, 3 and 4 of valid histories. To argue that the rules 9 and 10 cover the adding, removing and changing of actions each of these scenarios are examined:

- **Adding actions**: The rules of inconsistent cheating implies that the only consistent addition of actions are groups of actions representing executions, or groups of actions representing the execution of other events affecting the current event. Rule 10 handles this kind of violation.
- **Removing actions**: The rules of inconsistent cheating implies that the only consistent subtraction of actions are groups actions representing executions, or groups of actions representing the execution of other events affecting the current event. Rule 10 handles this kind of violation.
- **Changing actions**: Recall that an action consists of an ID, a timestamp, a counterpart ID, a counterpart timestamp, and an action type. The rules of inconsistent cheating implies that it is not possible to change the ID, the counterpart ID, or the action type consistently. This leaves the timestamp and counterpart timestamp as the only two values which can be altered consistently, since unoriginal timestamp values can still abide to Lamport's logical clocks. Therefore rule 9 covers this violation.

6.2.1 Detection Using DCR Graph Structure

Detecting and identifying consistently cheating processes is difficult, because it is not always possible to look at the history of a single event and determine if it is valid or not. It is now examined how the structure of the DCR graph and the placement of the malicious processes contribute to the ability to detect and identify consistent cheating. The type of relation has no influence on the cases and relations are therefore visualised as an arrow from event to event.

Case 1. In this case, the workflow contains an event that does not share relations with any other event, but may have relations to itself. If the event is hosted on a malicious process and it consistently cheats, there is no one to disagree with any timestamps on actions or oppose the amount of executions that have happened. Therefore, in workflows with such a structure, it is not possible to identify or even observe if any kind of consistent cheating has happened on this event.

Case 2. In this case, the workflow contains an event on a malicious process that has a relation to an event on a correct process as shown in figure 6.13. The malicious process can cheat consistently with the timestamps in two ways: It can change the timestamps of the actions corresponding to the relations with the counterpart, but in that case the two events will not agree on what happened and it is observable that that one of the processes hosting the event must cheat. If the timestamp changes are on actions not related to the good event then it is not possible to observe nor identify. An example of this is has been shown on figure 6.10.

In the same way, when looking at the amount of executions of the event on the malicious process, the malicious process might add more or remove some executions. In the case that it adds more executions, the event on the correct process will state that fewer executions happened, and vice versa. However, the malicious process is not able to disagree with the number of executions of the event on the correct process consistently.

Therefore it is possible to observe, but not identify, consistent cheating in this case.

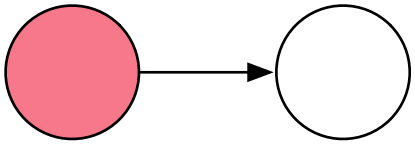


Figure 6.13: A workflow containing an event on a malicious process that has a relation to an event on a correct process.

Case 3. In this case, the workflow contains an event with a relation to an event on a malicious process as shown in figure 6.14. Similarly to case 2, timestamp changes are observable due to the same factors, but agreeing on exeuctions has another form. Instead of agreeing on the amount of executions of the event on the malicious process, the disagreement will be about the amount of executions of the event on the correct process. Furthermore, the malicious process is free to report any number of executions as in case one because no event can disagree on any outgoing actions.

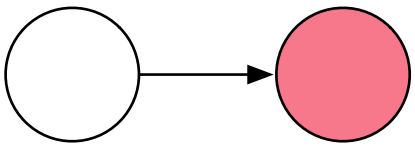


Figure 6.14: A workflow containing an event on a correct process that has a relation to an event on a malicious process.

Note that it is not possible to distinguish between disagreements as a result of cases 2 and 3, since they can create the exact same histories with the exact same disagreements.

Case 4. In this case, the workflow contains two events with relations to each other where one of the events is hosted on a malicious process, as shown on figure 6.15. The ability to observe and identify consistent cheating is similar to that of the cases 2 and 3 because the two events are able to disagree on both timestamps and the amount of executions of both events.

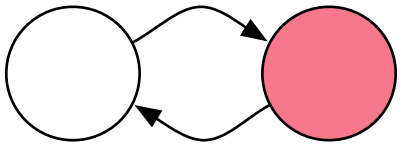


Figure 6.15: A workflow containing an event on a malicious process and an event on a correct process where the events have relations to each other.

Case 5. In this case, the workflow contains an event on a malicious process with a relation to another event on a malicious process as shown in figure 6.16. The two processes can work in collaboration to manipulate the histories of the two events.

If it is assumed that they work in collaboration, then they are able to change timestamps and the amount of executions in such a way that it is not observable because it is present in both the histories. However, if they do not change the timestamps or the amount of executions in collaboration, then they will disagree on the values and it is possible to observe that at least one of the processes are malicious. Because we assume the worst, it cannot be concluded that cheating is observable in this case.

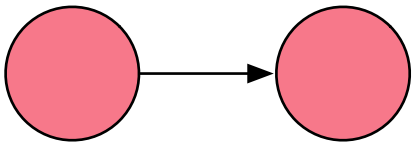


Figure 6.16: A workflow containing two events on malicious processes where one event has a relation to the other.

Case 6. In this case, the workflow contains two events hosted on malicious processes with relations to each other as shown in figure 6.17. This situation has the same properties as case 5.

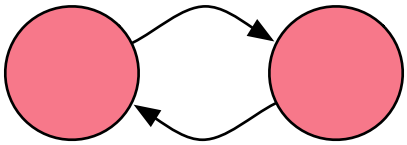


Figure 6.17: A workflow containing two events on malicious processes where the events has relations to each other.

It is not possible to identify which of the events are hosted on malicious processes in any of these cases. Furthermore, in cases 1, 5, and 6 it is not even possible to observe consistent cheating if the processes collaborate.

This creates some interesting questions about how one can try to identify which of the events are cheating when cheating is observed. One method of doing so could be to determine the malicious process by counting how many of the neighbouring events disagree with the event on the process. If the majority of the neighbours disagree, the process hosting the event will be marked as malicious. Unfortunately that method has drawbacks, since an event on a correct process with connections only to events on malicious processes could itself be marked as malicious.

If we require that in all neighbourhoods (the sets of neighbours of all events) of the DCR graph, with N events, a maximum of $N/2 - 1$ events are hosted on malicious processes, then an election would not be able to falsely accuse an event on a correct process of being malicious. This requirement is often needed in distributed systems that use elections, but traditionally it is only used to describe the amount of malicious nodes in the entire system and not in each neighbourhood. Therefore this requirement is more strict and perhaps more difficult to accomplish.

To make the situation substantially worse, even if this requirement is accomplished, a malicious process does not have to cheat with the actions related to a majority of neighbours of the event. An example could be a workflow with four events fully connected where one of the events is hosted on a malicious process. The malicious process could consistently cheat with one its relations, but keep the rest of its history valid. In that situation only one of the neighbours would vote against it.

As described in the example, it would be possible for the malicious process to trick an election into not having a majority against it. Therefore an election cannot conclude that a process *is* correct, which leaves the situation no better than the initial situation, where it was only possible to detect that at least one of the processes were malicious.

Another approach is to apply the solution to the Byzantine generals problem. Imagine that events are generals and their message exchanges are the decision value, then because at only two events know about a message exchange between them, using this approach contradicts the proven theorem that the Byzantine generals problem cannot be solved for systems with two processes.

Consistent Cheating	**Timestamp changes**	**Execution amt. of 1st event**	**Execution amt. of 2nd event**
Case 1	No	No	N/A
Case 2	Yes	Yes	No
Case 3	Yes	No	Yes
Case 4	Yes	Yes	Yes
Case 5	No	No	No
Case 6	No	No	No

Table 6.2: Table presenting whether or not it is possible to observe consistent cheating in the 6 cases.

From table 6.2 it becomes apparent that the structure of the workflow has a great influence on whether or not it is possible to observe consistent cheating - especially in cases where a lot of malicious processes are either interconnected or not connected to any other process at all. In these cases it can be difficult to observe any malicious activity. Furthermore, the cases also show that the more relations events on malicious processes have to events on correct processes and vice versa, the more it is possible to observe the malicious activity. Therefore a more connected workflow is preferred, because the probability for an event on a malicious process to be connected to an event on a correct process is higher.

6.3 Simulation

In order to validate that executions of events have only happened when the events were in an executable state, it is necessary examine each marking the workflow has transisted through. Because we only know the initial marking and the current marking of the workflow, but no markings in between, we cannot validate each execution without calculating the intermediate markings, because each execution of an event can change the marking of the workflow.

One approach to this problem is to *simulate* the executions of the workflow, that is given the initial marking, apply every execution from the order of execution, while validating that every marking, created by the previous executions, allows for the next execution to happen.

Because an order of execution can only be collapsed from a valid global history and valid global history can only be merged from valid local histories, it does not make sense to perform simulation if any kind of cheating has been observed.

In order to simulate an order of execution, it is required that the rules and the initial marking of the workflow is known. Furthermore, to simulate concurrent executions in a non-distributed environment, it is required to find a total order of execution from the partial order.

Since the order of execution states the happens-before relations of all executions, simulation must require that any execution must be applied to the marking, before any other execution it happens before. Recall that the topological order of a directed acyclic graph is a total ordering where for each edge from A to B, A before B. Since the topological ordering has the properties we need to execute concurrent executions in a non-distributed environment, we need to argue that any order of concurrent executions result in the same marking as applying the executions simultaneously.

As argued in chapter 5, concurrent executions can only be present in the order of execution if the executed events are independent. In [5], Debois et. al. conclude that any order of execution of independent events results in the same marking. Therefore any topological ordering of an order of execution is sufficient to simulate any ordering of concurrent events.

When a topological ordering of executions have been found, each execution is applied to the marking one at a time, resulting in a new marking.

If a marking does not allow for the next execution to happen, either the process hosting the executing event or the processes hosting any of its conditions are malicious. If the marking states that the executing event is not included, then the process hosting the executing event must be malicious. Alternatively, if the executing event is included in the marking, but at least one of its conditions are not fulfilled, this means that either the process hosting the condition returned incorrect information, or the process of the executing event disregarded the correct information of the condition. In these situations it is not possible to identify the cheating processes.

Furthermore, if the marking does not allow for the next execution to happen, multiple approaches can be taken. We have chosen to fail and return the invalid execution, because by choosing to apply the invalid execution or not to the marking, an assumption has been made about the rest of the executions.

Algorithm 5 The **Simulation** algorithm

```
function Simulation(orderOfExecution, DCRRules, initialMarking)
    orderedExecutions ← TopilogicalSort(orderOfExecution)        ▷ Returns a queue
    marking ← initialMarking
    while HasElements(orderedExecutions) do
        execution ← Pop(orderedExecutions)
        if not IsExecutable(execution.Event, state) then
            return Failure(execution)
        end if
        state ← ApplyExecution(execution.Event, DCRRules, state)
    end while
    return Success
end function
```

In algorithm 5 the simulation algorithm can be seen.

If every execution is simulated in a marking equal to the marking of the workflow at the time the execution happened, then we have shown that the order of execution *is* a run of the workflow. If it was not possible to simulate every execution, then the global history used to create the order of execution is not valid. Therefore the simulation algorithm must find violations of rule 3.

The simulation algorithm runs in time linear to the amount of executions plus the amount of happens before relations between the executions. This is because topological sort can be implemented with a time complexity of $\mathcal{O}(N + E)$ where N is the number of nodes (executions) and E is the number of edges (happens before relations), and the rest of the algorithm has a time complexity of $\mathcal{O}(N)$ because even though `IsExecutable` and `ApplyExecution` have time complexities of $\mathcal{O}(R)$ where R is the amount of relations in the workflow, the amount of executions will be the dominant factor most often. A run of algorithm 5 can be seen on figure 6.18.

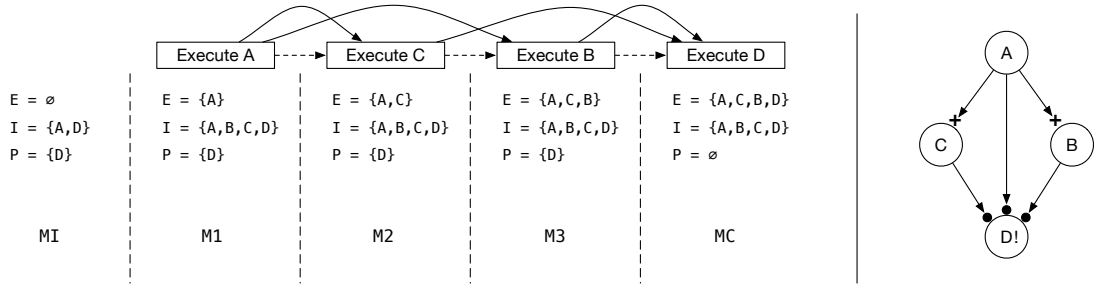


Figure 6.18: A simulation of a topological order of execution, based on the run of a workflow shown to the right. The marking is shown before and after each execution.

6.4 Correctness

In order to argue that inconsistent and consistent cheating cover all kinds of cheating, recall the four properties of a valid history:

1. Actions in the history are not invented, changed or removed.
2. The rules of the workflow are followed.
3. The history abides to serial equivalence.
4. The history is in a strict partial order.

If a history fulfils all properties, then it is valid. If a history violates property one, then a process must have cheated, either inconsistently or consistently. If at least one of the remaining properties are violated by the history, then a process must have cheated inconsistently. We have thus covered all kinds of cheating that can violate the properties of valid history.

We have shown that it is not possible to observe all kinds of consistent cheating, but given an ideal structure of a DCR graph together with a fitting distribution of events then it is possible for the different owners of the workflow, to validate each other and the histories their events provide.

Conclusion

This report has examined how it is possible to find a global order of execution from a run of a distributed DCR graph. Several algorithms are used in order to find this order of execution. By storing executions as actions performed by events as a totally ordered history on the individual events and using Lamport's logical clocks as timestamps to establish happens-before relations, it is possible to merge local histories together.

A combined history can be simplified to represent only the order of executions by collapsing actions of executions together to form single entities with the same happens-before relations, and finally using transitive reduction to find the minimum equivalent graph of the order of execution.

To reach distributed consensus on the order of execution, each event can examine the resulting order of execution and confirm that the local order execution of itself is preserved in the global order.

The observability and identifiability of malicious behaviour of processes in the DCR graph hosting events depend on both the type of cheating and the structure of the DCR graph. By applying validations on individual histories of events, pair validations between histories of two events and simulating the order of execution in the DCR graph, it is possible, in most cases, to observe malicious behaviour. If the amount of interconnectivity of the DCR graph is sparse, and malicious nodes are interconnected, then it is difficult to observe if some kind of cheating has happened at all.

Therefore it is up to the creators of workflows to create workflows with such structures and distribution of events across processes that all kinds of cheating are observable.

The time complexity of most algorithms have been analysed, and both from the analysis and from running the implementation it is clear that the transitive reduction algorithm used, is dominant factor in terms of computation time.

The transitive closure and the produce algorithm are examples of time consuming approaches to the problem that did not lead to sufficient results, satisfactory time complexities, or abilities to handle malicious processes. These algorithms are included in the report, to help the argumentation that the chosen algorithms are better approaches to solve the problem, given the distributed DCR graphs implementation which is the basis of the project.

Overall, algorithms have been described that solve the problem. Multiple areas could be extended upon in future projects, for example how malicious process are handled if they are identified, how much that can be said about a history if malicious behaviour is observed, with which structures of DCR graphs one could prevent malicious behaviour completely, if trust between processes could help in the identification of cheaters, in what cases it is impossible to find an order of execution at all, and finally, finding a peer-to-peer based gathering algorithm that would be able to handle malicious events.

Bibliography

[1] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*, chapter 2. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[2] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*, chapter 16. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[3] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*, chapter 15. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[4] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*, chapter 6. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[5] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. Concurrency and asynchrony in declarative workflows. In *Business Process Management*, pages 72–89. Springer, 2015.

[6] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In *FM 2015: Formal Methods*, pages 143–160. Springer, 2015.

[7] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Safe distribution of declarative processes. In *Software Engineering and Formal Methods*, pages 237–252. Springer, 2011.

[8] Thomas T Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. *arXiv preprint arXiv:1110.4161*, 2011.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[10] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[11] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[12] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014. USENIX Association.

[13] R. Sedgewick and K. Wayne. *Algorithms*, chapter 4. Pearson Education, 4th edition, 2011.