Transactions in Stack, Fork, and Join Composite Systems

Gustavo Alonso, Armin Feßler, Guy Pardon, and Hans-Jörg Schek

Institute of Information Systems
ETH Zentrum, CH-8092 Zürich, Switzerland
{alonso,fessler,pardon,schek}@inf.ethz.ch

Abstract. Middleware tools are generally used to glue together distributed, heterogeneous systems into a coherent composite whole. Unfortunately, there is no clear conceptual framework in which to reason about transactional correctness in such an environment. This paper is a first attempt at developing such framework. Unlike most existing systems, where concurrent executions are controlled by a centralized scheduler, we will assume that each element in the system has its own independent scheduler receiving input from the schedulers of other elements and producing output for the schedules of yet other elements in the system. In this paper we analyze basic configurations of such composite systems and develop correctness criteria for each case. Moreover, we also show how these ideas can be used to characterize and improve different transaction models such as distributed transactions, sagas, and federated database transactions.

1 Introduction

Composite systems consist of several components interconnected by middleware. Components provide services which are used as building blocks to define the services of other components as shown in fig. 1. This mode of operation is widely used, for instance, in TP-Monitors or CORBA based systems. To achieve true plug and play functionality, each component should have its own scheduler for concurrency control and recovery purposes. Unfortunately, most existing theory addresses only the case of schedulers with a single level of abstraction [6]. This is unnecessarily narrow: it hinders the autonomy of components and, by neglecting to take advantage of the available semantic information, also restricts the degree of parallelism. Except for open nested, multi-level transactions [14], almost no attention has been devoted to the case were several schedulers are interconnected with the output of one scheduler being used as the input to the next. In this paper, we address this problem by determining what information a scheduler must provide to another to guarantee global correctness while still preserving the autonomy of each scheduler. Based on multilevel [4,14], nested [9], and stack-composite [1] transactions we develop a theory that allows composite systems to be understood w.r.t. correct concurrent access. Of all possible configurations, we consider here only a few important cases: stacks, forks, and joins

Catriel Beeri, Peter Buneman (Eds.): ICDT'99, LNCS 1540, pp. 150–168, 1998.

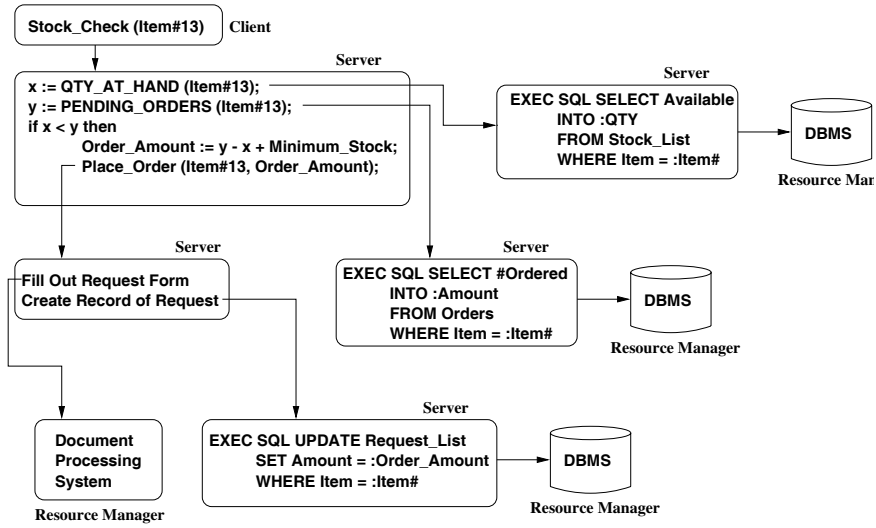

Fig. 1. An example transaction in a composite system

(see fig. 2) and simple combinations of them. The idea is to use these as building blocks that can later help us to understand and model more complex cases. We see the contribution of our paper in providing a formal basis for correctness in these architectures. In addition, we show how several classical problems of transaction management can be expressed and explained in a coherent manner using the proposed framework, without having to resort to separate models for each case. In particular, we prove this by showing that traditional distributed transactions, sagas, and federated databases (global and local transactions) are special cases of our model. The paper is organized as follows. Section 2 presents the transaction model and introduces conflict consistency as our basic correctness criterion. Section 3 discusses correct concurrent executions in stacks, forks, and join schedules and introduces a simple combination of forks and joins. Section 4 discusses distributed transactions, sagas and federated transactions within the framework of composite schedulers. Finally, section 5 concludes the paper with a brief summary of the results and future work.

2 Conflict Consistency

In this section, we present our basic correctness criteria. These serve as a necessary preparation for the rest of the paper, where it becomes obvious how operations of a scheduler act as transactions in other schedulers. In what follows, we assume familiarity with concurrency control theory [5].

2.1 The New Model

When executing transactions, a scheduler restricts parallelism because it must, first, observe the order constraints between the operations of each transaction and, second, impose order constraints between conflicting operations of different transactions. The restriction in parallelism occurs because, in a conventional scheduler, ordered operations are executed sequentially. As shown in [1], this is too restrictive. It is often possible to parallelize concurrent operations even when they conflict as long as the overall result is the same as if they were executed sequentially. This requires to relax some of the ordering requirements of traditional schedulers. In addition, when several schedulers are involved, a mechanism is needed to specify to a scheduler what is a correct execution from the point of view of the invoking scheduler. For these two purposes we use the notion of *weak* and *strong* orders (assumed to be transitively closed):

Definition 1 (Strong and Weak Order:). *Let A and B denote any tasks (actions, transactions).*

- *Sequential (strong) order: $A \ll B$, A has to complete before B starts.*
- *Unrestricted parallel execution: $A \| B$, A and B can execute concurrently equivalent to any order, i.e., $A \ll B$ or $B \ll A$.*
- *Restricted parallel (weak) order: $A < B$, A and B can be executed concurrently but the net effect must be equivalent to executing $A \ll B$.* □

From here, and independently of the notion of equivalence used, it follows that turning a weak order into a strong one leads to correct execution since this implies sequential execution. In fact, traditional flat schedulers with only one level of abstraction, do just this, if two tasks conflict they impose a strong order between them. When moving to a composite system with several schedulers, it is often possible to impose a weak order instead of a strong one, thereby increasing the possibility of parallelizing operations. Thus, the aim will be to impose either no orders or only weak orders while still preserving global correctness and characterize the exchange of information between schedulers in terms of these orders. Note that in our model, weak and strong orders are *requirements and not observed (temporal) orders*. A strong order implies a temporal one, but not the other way round. In figure 3, t_3 is weakly input-ordered before t_1, and the serialisation order is according to it. Therefore, the execution is correct. However, t_1 is temporally ordered before t_3 (the execution order is indicated by the position: left comes first). This shows that the temporal order may be irrelevant. In other words, order preservation as postulated in [3] is not required in our model. With these ideas, a transaction is now defined as follows. Let $\widehat{O}$ be the set of all operations of a scheduler with which transactions can be formed.

Definition 2 (Transaction). *A* transaction*, t, is a triple (O_t, $<_t$, $\ll_t$), where O_t is a set of* operations *taken from $\widehat{O}$, $<_t$ is a partial order on O_t termed the* weak (intra-)transaction order*, and $\ll_t$ is a partial order on O_t termed the* strong (intra-)transaction order*. For consistency we require $\ll_t \subseteq <_t$.* □

Since now we have operations being executed at different schedulers, a modified notion of conflict is needed. Let $CON \subseteq \widehat{O} \times \widehat{O}$ be a conflict predicate that expresses whether two operation invocations commute. We say that two operations, o and o', commute if there is no difference in return values of the sequence $\alpha\ o\ o'\ \beta$ compared to $\alpha\ o'\ o\ \beta$ for all sequences α and β with elements from $\widehat{O}$ [13]. Therefore, commutativity is not an absolute property but is relative to the given set of (allowed) operation invocations. From here, we say that two operations conflict if they do not commute, which also includes the case when it is unknown whether they commute or not. In practice, in a composite system, two operations conflict if there is a potential flow of information between them. With this, now we have all the necessary elements to formally define a scheduler:

Definition 3 (Schedule). *A* schedule *S is a five-tuple $(T, \rightarrow, \mapsto, <, \ll)$, where:*

- *T is a set of transactions.*
  Let O denote the set of all operations of T's transactions, i.e., $O = \bigcup_{t \in T} O_t$.
- *$\rightarrow$ and $\mapsto$ are the* weak *and* strong input orders*, partial orders over T with $\mapsto \subseteq \rightarrow$.*
- *$<$ and $\ll$ are the* weak *and* strong output orders*, partial orders over O such that:*
  1. $\forall t, t' \in T, t \neq t',\ \forall o \in O_t, \forall o' \in O_{t'}, CON(o, o') :$
     (a) $(t \rightarrow t') \Rightarrow (o < o')$
     (b) $(t' \rightarrow t) \Rightarrow (o' < o)$
     (c) *otherwise:* $(o < o') \vee (o' < o)$
  2. (a) $\forall t \in T,\ \forall o, o' \in O_t : (o <_t o') \Rightarrow (o < o')$,
     (b) $\forall t \in T,\ \forall o, o' \in O_t : (o \ll_t o') \Rightarrow (o \ll o')$,
  3. *Whenever $t \mapsto t'$, then $\forall o \in O_t, \forall o' \in O_{t'} : o \ll o'$,*
  4. $\ll \subseteq <$. □

According to point 1, a scheduler must weakly order every pair of conflicting operations without contradicting the weak input order, if any, between the parent transactions (otherwise, as we will see below, a cycle would immediately appear). Point 2 ensures well-formedness, that is, all weak and strong transaction orders are contained in the weak and strong output orders, respectively. Point 3 propagates the strong input order from the transactions to their operations, thereby separating the execution tree of strongly ordered transactions. Point 4 guarantees that the output orders of a scheduler are consistent between them. It is not stated explicitly that if at least one of two weakly output ordered operations is a leaf, then they are also strongly ordered. This is an evident requirement.

2.2 Conflict Consistency

The distinction between the two orderings requires to modify the traditional notion of correctness. We will assume, as usual, that a transaction executed in isolation is correct.

Definition 4 (Serial Schedule).
A schedule S is serial *if $\mapsto_S$ is a total order, i.e., $\forall t, t' \in T : (t \mapsto t') \vee (t' \mapsto t)$.* □

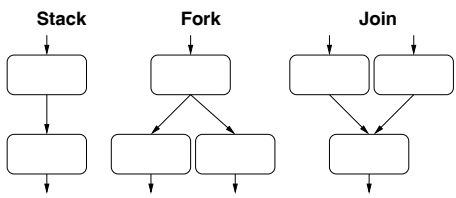

Fig. 2. Coupling modes between schedulers

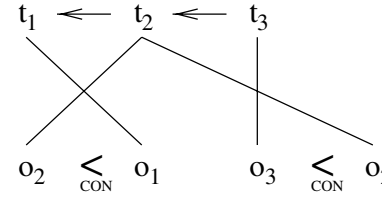

Fig. 3. Example for non-relevant temporal execution order

Note that serial schedule does not mean that all operations are executed serially. Operations within one transaction can be executed in parallel. In a serial schedule we have $\mapsto = \rightarrow$. Following the classical approach, we can now define the new correctness criterion, called *conflict consistency*.

Definition 5 (Conflict Consistency (CC)).
A schedule S is conflict consistent *if there is a serial schedule S_{ser}, whose strong input and weak output order contain the weak input and output order of S, resp., i.e.,* $(\rightarrow_S \subseteq \mapsto_{S_{ser}}) \wedge (<_S \subseteq <_{S_{ser}})$. □

The name *conflict consistency* expresses that the order of all *conflicting* operations must be consistent with the weak input order, i.e., the serialisation order must not contradict the weak input order. This can be further formalized as follows:

Definition 6 (Serialisation Graph $\hookrightarrow$). *Given a schedule S, its* serialisation graph*, denoted $\hookrightarrow$, is a transitive irreflexive binary relation on $T \times T$, in which $t \hookrightarrow t'$ is contained if $t \neq t'$ and $\exists o \in O_t,\ \exists o' \in O_{t'} :\ CON(o, o') \wedge (o < o')$* □

Theorem 1. *A schedule S is conflict consistent iff the union of its weak input order and its serialisation graph $(\hookrightarrow_S \cup \rightarrow_S)$ is acyclic.* □

This can be proven by constructing a total order containing the union of weak order and serialisation graph, thereby defining a serial schedule with the same weak output order (see appendix). This theorem does not take explicitly into account the strong input order since it is contained in the weak one (see def. 3).

Compared to [1] we simplified the definitions: we dropped the notion of equivalent schedules and we included a clearer presentation of order constraints.

2.3 Recovery

For a formal treatment of recovery, we use the unified theory for concurrency control and recovery [2,13]. This theory is based on an expanded schedule which is used to represent the transaction's recovery operations explicitly, i.e., in every schedule each abort is replaced by the corresponding inverse operations ("undos") in the appropriate order. It has been shown that ordering commits in

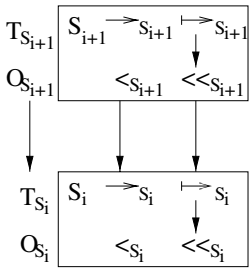

Fig. 4. Stack.

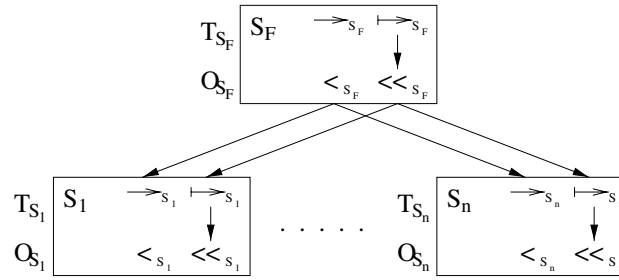

Fig. 5. Fork.

the order of conflicting operations and aborts in the opposite direction (SOT in [13]) leads to correct concurrency control and recovery. In order to achieve this, every scheduler must automatically generate the appropriate invocations of inverse operations properly ordered in case of a failure. If inverse operations are not available we will assume that a scheduler defers the commit of all operations that do not have inverses. A more comprehensive treatment of recovery is beyond the scope of this paper. We will concentrate here on concurrency control.

3 Composite Systems

In a composite system, every server has a scheduler S and provides a set $\widehat{O}_S$ of operation invocations to be used to build transactions (the server's services), i.e., an operation of a scheduler can be and often will be a transaction of another one. Every scheduler S has a commutativity specification expressed by the conflict predicate CON_S. Every scheduler works locally ensuring correctness with respect to its (local) CON_S. The question to address is how to guarantee global correctness in such a scenario and what information is needed at each scheduler to guarantee global correctness.

3.1 Stack Schedules

Stack schedulers take the output of one scheduler and use it directly as input to the next (see fig. 4). Transactions can have different depths, but operations at the same level are always processed by the same scheduler. This structure is a generalization of multi-level and nested transactions [10,4,3,14] and it can be found in the internal structure of many systems. The notion of stack used in this paper is taken from [1]. Here, in addition, we prove correctness of stack schedules.

Definition 7 (Stack Schedule (SS)).
SS, an n-level stack schedule*, consists of* n *schedules* $S_1, \ldots, S_n$*, such that, for* $1 < i \leq n$*:*

- $T_{S_{i-1}} = O_{S_i}$
- $\rightarrow_{S_{i-1}} = <_{S_i}$
- $\mapsto_{S_{i-1}} = \ll_{S_i}$

□

This definition states that operations and their output orders are transactions and input orders of the next lower schedule. The important aspect of definition 3 together with this definition is the way in which orders are propagated. Only the strong ordering is automatically propagated to all levels, thereby ensuring that if two transactions are strongly ordered, their execution trees will not be interleaved at any lower level. The weak order is propagated from one level to the next only if the operations involved conflict. If there is no weak input order among the parents, but two children operations conflict, the scheduler introduces a weak output order.

Definition 8 (Stack Conflict Consistency (SCC)). *An n-level stack schedule SS is* stack conflict consistent *iff each individual schedule S_i in SS is conflict consistent, for* $1 \leq i \leq n$. □

Theorem 2. *A stack schedule is correct if it is SCC.* □

We prove this theorem by constructing a serial execution of all transactions of all levels, i.e., on all levels all transactions are strongly ordered (see appendix).

Notice that conflict consistency requires the existence of a serial schedule, where each transaction is executed serially, but not necessarily its operations. The advantage of SCC is that as long as each level independently enforces CC, the overall execution is correct. The weak order constraint and its careful propagation through the stack is important because often we do not know whether operations conflict or not. In these cases, one has to be careful and assume that they conflict. In contrast to existing multilevel transaction models [3,14], such a weak order constraint is irrelevant if there is no actual conflict at the next level. In [1], it is shown that SCC is a larger class than *order preserving serialisability* [3] and *level-by-level serialisability* [14], the two existing comparable criteria. Because we allow weak orders within transactions, SCC is also larger than multi-level-serialisability (MLSR in [14]).

3.2 Fork Schedules

In a fork (fig. 5), the output of a schedule is used as input to several other schedulers. Each pair top-level/lower-level scheduler can be seen as a stack and, indeed, it will follow the rules defined for stack schedulers. More formally:

Definition 9 (Fork Schedule (FS)). *A* fork schedule FS *consists of (n+1) schedules $S_F, S_1, \ldots, S_n$, such that:*

1. $O_{S_F} = \bigcup_{i=1}^{n} T_{S_i}$
2. $\forall i \in \{1, \ldots, n\} : \forall t, t' \in T_{S_i} : \begin{cases} t <_{S_F} t' \Rightarrow t \rightarrow_{S_i} t' \\ t \ll_{S_F} t' \Rightarrow t \mapsto_{S_i} t' \end{cases}$
3. $\forall (o_i, o_j), o_i \in O_{S_i}, o_j \in O_{S_j}, i \neq j$, *we assume* o_i *and* o_j *commute.* □

The output orders $<_{S_F}$ and $\ll_{S_F}$ of S_F are passed to the related component S_i as input orders. Note that every operation in S_F is sent to only one scheduler

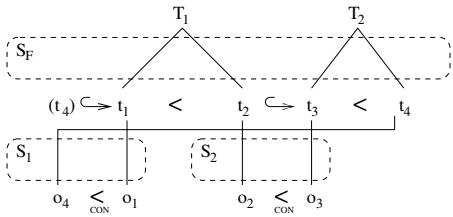

Fig. 6. Pure fork.

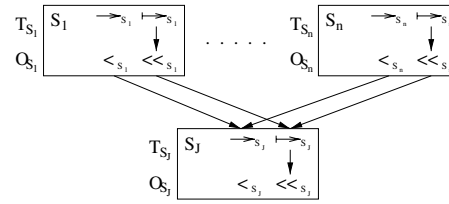

Fig. 7. Join.

S_i as a transaction. In "pure" forks we assume that invocation hierarchies are completely separated. No information flows from S_i to S_j or vice versa. This is why we assume that operations commute (point 3). Therefore, every weak order between two operations going to different schedulers S_i and S_J disappears. However, every weak order between two operations going to the same scheduler S_i is transformed to an input order in S_i, so this scheduler takes care of it. For instance, the execution in figure 6 is a correct pure fork. Pure forks are very common in practice, as shown in figure 1.

Definition 10 (Fork Conflict Consistency (FCC)). *A fork schedule FS is* fork conflict consistent (FCC)*, if the schedule S_F is conflict consistent and $\bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ is acyclic.* □

Theorem 3. *An execution in a fork schedule is correct iff it is FCC.* □

This can be proven by constructing an equivalent stack schedule (see appendix). FCC is a global criterion. Since we want to check for correctness locally, each scheduler should be able to decide independently if the schedule is correct or not. This can be done as follows:

Theorem 4 (Criterion for Fork Conflict Consistency). *A fork schedule FS is fork conflict consistent, iff each of the schedules $S_F, S_1, \ldots, S_n$ is conflict consistent.* □

This can be proven by dividing $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ into subgraphs of the different schedules and considering their connections to each other (see appendix).

Forks (fig. 6) are a straightforward extension of stacks. The criterion for forks becomes more complicated when we allow that operation invocation hierarchies are not separated, i.e., when after the fork, it is possible to have a join schedule. This case is discussed in the next section.

3.3 Join Schedules

Join schedulers[1] entail several top schedulers whose outputs go to a common, single low-level scheduler (figure 7), called S_J. Joins are common at the resource

[1] Join schedules are not to be confused with join-transactions introduced in [11]

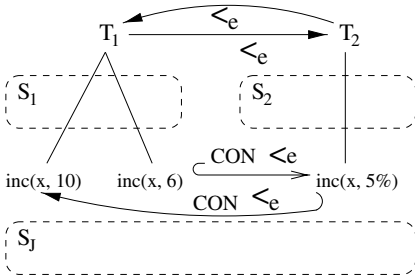

Fig. 8. Join with two transactions.

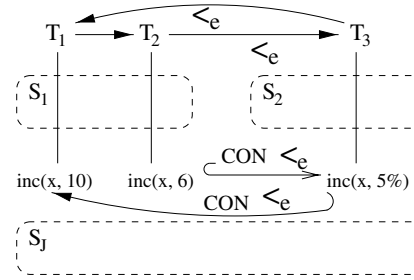

Fig. 9. Join with three transactions.

manager level, where a single resource manager is being accessed by several servers simultaneously. Again, each pair top-level/low-level schedule behaves like a stack, but care must be taken to avoid inconsistencies. For this purpose, conflicts must be assumed between operations of different higher level schedulers. The formal definition of join schedule is a straightforward extension of that of stacks: the output of the top-level schedulers (operations, and weak and strong orders) is used as the input to the lower-level scheduler (abusing notation, also called *join* schedule). Thus,

Definition 11 (Join Schedule (JS)). *A* join schedule JS *consists of (n+1) schedules* $S_J, S_1, \ldots, S_n$*, such that:*

- $\bigcup_{i=1}^{n} O_{S_i} = T_{S_J}$
- $\forall i \in \{1, \ldots, n\} : \forall t, t' \in O_{S_i} : \begin{cases} t <_{S_i} t' \Rightarrow t \rightarrow_{S_J} t' \\ t \ll_{S_i} t' \Rightarrow t \mapsto_{S_J} t' \end{cases}$ □

Intuitively, since the lower level schedule follows the input orders provided, each pair top-level/lower-level scheduler behaves like a stack. From a correctness point of view, it remains to be determined whether and how to interleave transactions from different schedulers. The problems to avoid are illustrated in figures 8 and 9.

Assume two users, each one operating on one of two top-level schedulers in a join. In figure 8 the information flow from T_1 to T_2 and back can be detected neither by S_J nor S_1 nor S_2. This problem could be solved if, for instance, S_J would build the serialisation graph based on the root transactions, which would result in a cycle. In figure 9, however, this does not help because the information flows from T_2 to T_3 to T_1. The cycle can only be detected when considering the weak input order between T_1 and T_2.

In other words, assume T_1 and T_2 were directly given to S_J and not to S_1 and S_2 as before. Then S_J would directly reject the execution because it is not CC. To capture these situations we introduce the *ghost-graph*. Let $Act(T)$ represent the children of transaction T at all levels below that in which T appears.

Definition 12 (Ghost-Graph for Join Schedules($\lesssim_{JS}$)).
$\forall (T, T')$ *with* $T \in T_{S_i}, T' \in T_{S_j}, i \neq j$ *the* ghost-graph $\lesssim_{JS}$ *is defined as:*
$T \lesssim_{JS} T'$ *if there are children* t, t' *of* T, T'*, resp., with* $t, t' \in T_{S_J}$ *and* $t \hookrightarrow_{S_J} t'$. □

Definition 13 (Join Conflict Consistency (JCC)). *A join schedule JS is* join conflict consistent, *if the schedule S_J is conflict consistent and $\lesssim_{JS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ is acyclic.* □

This definition matches the intuition that a join schedule should be considered correct if the whole stack schedule built from all partial schedules – together with the implicit (ghost) orderings between different schedules – is SCC.

Theorem 5. *An execution in a join schedule is correct iff it is JCC.* □

This can be proven by constructing an equivalent stack schedule that is SCC (see appendix). As pointed out above, the idea is that each scheduler will be able to make decisions locally. The criterion JCC, however, is a global property that cannot be enforced locally. Fortunately we can artificially generate weak input orders between all sub-transactions of transactions that come from different schedulers. The weak input order expresses the fact that we must assume a conflict between operations of such transactions. Remember that the weak input order generally stems from output orderings of conflicting operations at a level above. Since there is no common level above, we do not know about the commutativity of such operations and, therefore, must assume a conflict. This idea is formalized as follows:

Definition 14 (Completed Join Schedule (CJS)). *A* completed join schedule CJS *is a JS with additional input order compatible with:*
$\forall S_i, S_j, i \neq j : (\forall t \in T_{S_i}, \forall t' \in T_{S_j} : t \rightarrow t') \vee (\forall t \in T_{S_i}, \forall t' \in T_{S_j} : t' \rightarrow t)$ □

Theorem 6 (Criterion for JCC). *A completed join schedule CJS is JCC, if each of the schedules $S_J, S_1, \ldots, S_n$ is conflict consistent.* □

This locally testable criterion for JCC can be proven by reducing the ghost-graph to weak input orders between transactions of different schedules (see appendix).

From here, if each transaction that arrives at S_J contains the information about its parent scheduler, S_J can impose the additional weak input orderings. Then S_J can decide locally about correctness of the join schedule. In figures 8 and 9, S_J would impose an order from `inc(x,10)` and `inc(x,6)` to `inc(x,5%)` or vice versa. So, a cycle in the union of weak input order and serialisation graph of S_J would be detected.

Note that there exist JCC join schedules that cannot be transformed into a correct CJS. Counterexamples are join schedules whose ghost-graph builds a cycle "between schedules" but not between transactions. CJS is a purely static notion and cannot be used in practice for dynamic scheduling. Dynamic scheduling in these scenarios is a complex problem which can be addressed by adding further restrictions to the execution but which is beyond the scope of this paper.

3.4 FDBS-Schedules

The FDBS-schedule will be used later to describe federated databases. A federated database can be described by a fork schedule with additional schedules at

the same level as S_F, one virtual schedule S_{Li} for each local transaction t_{Li} (see figure 11):

Definition 15 (FDBS-Schedule (FDS)). *An* FDBS-schedule FDS *consists of (n+l+1) schedules $S_F, S_{L1}, \ldots, S_{Ll}, S_{J1}, \ldots, S_{Jn}$, such that:*

1. $O_{S_F} \cup \bigcup_{i=1}^{l} O_{S_{Li}} = \bigcup_{i=1}^{n} T_{S_{Ji}}$
2. $\forall i \in \{1, \ldots, l\} : T_{S_{Li}} = \{t_{Li}\}, O_{T_{Li}} = \{o_{Li}\}$
3. $\forall i \in \{1, \ldots, n\} : \forall t, t' \in T_{S_{Ji}} : \begin{cases} t <_{S_F} t' \Rightarrow t \rightarrow_{S_{Ji}} t' \\ t \ll_{S_F} t' \Rightarrow t \mapsto_{S_{Ji}} t' \end{cases}$ □

FDBS-schedules have the same problems as a join. Thus, to define correctness, we use a similar notion:

Definition 16 (Ghost-Graph for FDBS-Schedules($\lesssim_{FDS}$)).
$\forall (T, T')$ *with* $T \in T_S, T' \in T_{S'}, S, S' \in \{S_F, S_{L1}, \ldots, S_{Ll}\}, S \neq S'$ *the* ghost-graph $\lesssim_{FDS}$ *is defined as:*
$T \lesssim_{FDS} T'$ *if there are children* t, t' *of* T, T'*, resp., with* $t, t' \in T_{S_{Ji}} (i \in \{1, \ldots, l\})$ *and* $t \hookrightarrow_{S_{Ji}} t'$. □

Not surprisingly, FDBS conflict consistency can be defined as join conflict consistency.

Definition 17 (FDBS Conflict Consistency (FDCC)). *An FDBS-schedule FDS is* FDBS conflict consistent*, if the schedules $S_{J1} \ldots S_{Jn}$ are conflict consistent and $\lesssim \cup (\hookrightarrow_{S_F} \cup \rightarrow_{S_F})$ is acyclic.* □

Theorem 7. *An execution in an FDBS-schedule is correct iff it is FDCC.* □

This can be proven by constructing an equivalent stack schedule (see appendix). Note that serialisation graphs and weak input orders can not appear in the "local schedules" S_{Li} since they are only virtual schedules. Thus, since FDCC cannot be applied locally we have to seek another criterion. For this, we define:

Definition 18 (Completed FDBS-Schedule (CFDS)). *A* completed FDBS-schedule CFDS *is an FDS with additional conflicts:* $CON_{S_F} := CON_{S_F} \cup \{(o, o') \mid o \in O_t, o' \in O_{t'}, t \neq t', t, t' \in T_{S_F}, \exists i : (o, o' \in T_{S_{Ji}})\}$
Note that this implies that S_F has to weakly order these extra conflicts, as required by its CC property. □

Now, a CFDS is correct (FDCC), if the following holds:

Theorem 8 (Criterion for FDCC). *A completed FDBS-schedule CFDS is FDCC, if*
(1) each of the schedules $S_F, S_{L1} \ldots S_{Ll}, S_{J1} \ldots S_{Jn}$ is conflict consistent and
(2) if $\nexists t, t' \in O_T, T \in T_{S_F} : (t, t' \in T_{S_{Jj}} (j \in \{1 \ldots n\}, \exists t_{L1} \ldots t_{Lk} \in \bigcup_{i=1}^{l} O_{S_{Li}} :$
$(t \hookrightarrow_{S_{Jj}} t_{L1} \hookrightarrow_{S_{Jj}} \ldots \hookrightarrow_{S_{Jj}} t_{Lk} \hookrightarrow_{S_{Jj}} t'))$. □

The last condition means that in any S_{Jj} no local transactions must be serialised between global (sub-)transactions of the same parent (=transaction in S_F). We prove this theorem by assuming a cycle in the union of input, serialisation and ghost-order and showing a contradiction to CC of S_F (see appendix).

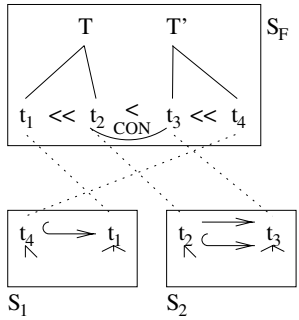

Fig. 10. An MDBS with partially non-conflicting subtransactions.

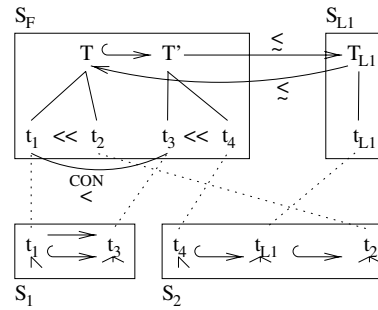

Fig. 11. FDBS: example.

4 Existing Composite Transaction Models

In this section we describe several transaction models using the ideas above. In particular, we consider distributed transactions, sagas, and federated database transactions. These models are two-level models: transactions consist of subtransactions considered as operations at the top-level, and executed as transactions at the lowest level. Top-level transactions are called global transactions, operations in global transactions are global subtransactions and transactions circumventing the global layer are called local transactions.

4.1 Distributed (Multi-database) Transactions

The model of interest here is often referred to as multidatabase system (MDBS). Following the usual terminology, $T_i (i = 1, 2, ..., n)$ are global transactions. Each global transaction is decomposed into global subtransactions, t_{ij}, encompassing all operations to be executed at the component databases, $S_j (j = 1, ..., m)$. For FS, the set T_{S_F} is the set of global transactions and O_{S_F} is the union of all subtransactions t_{ij}. For the weak and strong input order we set $\rightarrow_{S_F} = \mapsto_{S_F} = \emptyset$. For the bottom schedulers S_j, the set T_{S_j} of transactions are the subtransactions t_{ij}. O_{S_j} is the set of operations at S_j to be executed for all t_{ij}. In the classical treatment of distributed transactions nothing is known about the commutativity of subtransactions. This is expressed in the scheduler S_F by assuming a conflict between every pair of subtransactions: t_{ij} conflicts with $t_{rs} \Leftrightarrow i \neq r \wedge j = s$. Therefore S_F orders all such pairs the same way: For all $i_1, i_2 \in \{1, 2, ...n\}$: If $t_{i_1 j_1} < t_{i_2 j_1} \Rightarrow t_{i_1 j_2} < t_{i_2 j_2}$. Otherwise $\hookrightarrow_{S_F}$ would by cyclic. Therefore the weak input order imposed to all component schedulers is the same for all S_j. According to FCC all S_j are CC. From there FCC - in this special MDB setting - requires that all serialisation orders be the same.

This result clearly is not surprising and corresponds to the usual understanding. The advantage of our treatment is that several points become clear. First,

without further semantic knowledge the technique cannot be improved. Second, and more importantly, as soon as the S_F scheduler has information about the commutativity of the subtransactions, we can drop the related weak order constraint imposed to the corresponding S_j scheduler. Such an example is shown in figure 10, where there is no conflict between t_1 and t_4, thus allowing the serialisation order from t_4 to t_1.

Now, with respect to recovery, usually we do not assume to have inverses for global subtransactions. Therefore S_F must defer the commit of every subtransaction t_{ij} up to the commit of the global transaction Ti. This calls for atomic commitment as it is known. However, we can easily prove that, as soon as there are inverses t_{ij}^{-1} known to some t_{ij}, the scheduler can early commit such t_{ij} and, in case of failure, do recovery at the S_F level.

4.2 Sagas

A saga [7] T_i is a logical unit of work and shall be executed as a whole or not at all. It consists of a (partially) strongly ordered set of (sub-)transactions t_{ij}. Each transaction has an inverse t_{ij}^{-1} for compensation in case of recovery. With respect to concurrency, a saga does allow any kind of interleaving w.r.t. other concurrent sagas.

Let us describe sagas in a two-level stack or fork schedule consisting of a top-level schedule and one or more bottom-level schedules. The top-level scheduler assumes commutativity of all transaction pairs $(t_{ij}, t_{rs}), i \neq r$. Therefore no weak output order is generated and imposed as input to the bottom schedulers. No constraints are imposed over the serialisation orders of the subtransactions executed in S_j.

Again, this result is not surprising. However, as before we can easily incorporate the fact that not all transactions commute and not all transactions have inverses.

In comparison with distributed transactions and in summing up we have the following observation: *Distributed transactions correspond to a fork schedule where the transactions of different sagas conflict and no inverse transaction is known. Sagas correspond to the case where the transactions of different sagas commute and every transaction has an inverse.* From here, it is clear that our general fork schedule encompasses all cases "in between" these two extremes, i.e., if some transactions commute or if some transactions have inverses.

4.3 Federated Transactions

Federated transactions are a generalization of MDB transactions in that local transactions t_{Lk} circumvent the top-level scheduler and directly enter the bottom scheduler S_{Jj} (fig. 11). We have modelled this by a fork schedule with an (artificial) additional schedule S_{Lk} at the same level as S_F for every local transaction T_{Lk}, which we called an FDBS-Schedule (section 3.4). A correct FDBS is ensured by having the FDCC property, as mentioned earlier. Given

this property, we easily relate special results obtained in the context of multi-level transactions [12]. There a "dynamic" conflict relation to be used by the top-level scheduler was introduced in order to capture indirect conflicts between local transactions and global subtransactions. This corresponds to the additional edges of the ghost-graph in S_F of definition 16.

As a practical method, [12] have proposed to introduce two kinds of commits: a commit with respect to other global subtransactions at the end of every subtransaction and a commit with respect to local transactions at the end of the global transaction. As a simple method for an implementation with locking, it was proposed to have "retained" locks, i.e. locks that are kept until the end of the global transaction to shield against local transactions. Such locks are not visible to other global subtransactions. The effect of this locking scheme is that no ghost order can arise between active transactions. As a result, the ghost order represents a temporal relationship reflecting the fact that two transactions were executed serially with respect to each other, making cycles impossible.

The FDCC property also allows a simple explanation of the ticket method for federated databases [8]. In this method, all global transactions have at most one operation at each local (join) site, hence there is no need for the join to check whether a local transaction is serialized between two global subtransactions of the same root. Furthermore, a global transaction must read and increase the value of a counter at each site. A global commit is done only if there is no cycle based on the ticket orders between different global transactions. Because there is no input order in traditional schedulers, this method has to rely on the ticket values to determine the serialization order at the local databases. The assumption that every pair of operations conflict (see definition 18) is translated here in the fact that a value has to be updated by each transaction, forcing all of them to conflict.

5 Conclusion

In this paper, we have developed a framework to reason about correctness in composite systems. Our main goal has been to allow individual components to decide locally while still ensuring global correctness and to determine what information needs to be provided to each scheduler in order to do so. We use the notion of conflict consistency to restrict correct executions to those consistent with the information passed to a scheduler (weak and strong input orders). Based on conflict consistency, correctness criteria for several configurations (stack, fork, and join) have been developed. These criteria characterize all correct and only correct executions. In particular, if the composite system has a tree configuration (no joins), it suffices to enforce conflict consistency locally to obtain global correctness. If joins are involved, however, additional restrictions are necessary since the criteria provided are based on non-local information (the ghost graph). As the example with federated transactions shows, it is still possible to design dynamic criteria for configurations containing joins in spite of the static nature of the criterion for joins. As an additional contribution, we have described

known transaction mechanisms (distributed transactions, sagas, and federated transactions) as special cases of the composite framework and from there, we have demonstrated how more general mechanisms can be proven correct.

Future work involves dynamic schedulers including recovery within the composite framework and exploring more complex configurations, always with the trade-off in mind between passing global information and deciding locally.

References

1. G. Alonso, S. Blott, A. Fessler, and H.-J. Schek. Correctness and parallelism in composite systems. In *Proc. of the 16th Symp. on Principles of Database Systems (PODS'97)*, Tucson, Arizona, May 1997.
2. G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H.-J. Schek, and G. Weikum. Unifying concurrency control and recovery of transactions. *Information Systems*, 19(1):101–115, Jan. 1994. An extended abstract of this paper appeared in the *Proceedings of the 4th International Conference on Extending Database Technology*, EDBT'94. March, 1994, pages 123-130.
3. C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, Apr. 1989.
4. C. Beeri, H.-J. Schek, and G. Weikum. Multilevel transaction management: Theoretical art or practical need? In *Proceedings of the International Conference on Extending Database Technology*, number 303 in Lecture Notes in Computer Science, pages 134–154, Venice, Italy, Mar. 1988. Springer-Verlag, New York.
5. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.
6. Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, (1):181–239, 1992.
7. H. García-Molina and K. Salem. Sagas. In *Proc. 1987 SIGMOD International Conference on Management of Data*, pages 249–259, May 1987.
8. D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. Using tickets to enforce the serializability of multidabase transactions. *IEEE Knowl. and Data Engineering*, Feb. 1994.
9. J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts, MIT Press, 1981.
10. J. E. B. Moss, N. D. Griffith, and M. H. Graham. Abstraction in recovery management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 72–83, 1986.
11. C. Pu, G. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th International Conference on Very Large Databases, Los Angeles*, pages 26–37, 1988.
12. H.-J. Schek, G. Weikum, and W. Schaad. A Multi-Level Transaction Approach to Federated Transaction Management. In *Proceedings of the International Workshop on Interoperability in Multi-database Systems, Research Issues in Data Engineering (IEEE-RIDE-IMS)*, Kyoto, Apr. 1991.
13. R. Vingralek, H. Hasse, Y. Breitbart, and H.-J. Schek. Unifying concurrency control and recovery of transactions with semantically rich operations. *Journal of Theoretical Computer Science*, 1998.
14. G. Weikum. Principles and Realisation Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132, March 1991.

6 Appendix

This section contains the proofs to the theorems in the paper.

Theorem 1. *A schedule S is conflict consistent iff the union of its weak input order and its serialisation graph $(\hookrightarrow_S \cup \rightarrow_S)$ is acyclic.* □

This theorem was proven in [1] where we used slightly modified definitions. The idea is to construct a total order containing the union of weak order and serialisation graph, and to find a serial schedule with the same weak output order.

Theorem 2. *A stack schedule is correct if it is SCC.*
Proof. *We construct by induction a serial execution of all transactions of all levels, i.e., on all levels all transactions are strongly ordered:*
We can formulate the induction in two steps:
(1) Whenever a serial schedule S'_{i-1} can be constructed which orders conflicting operations as $<_{S_{i-1}}$ and serialises the transactions in T_{i-1} as they are ordered in $<_{S_i}$ the following holds:
(2) We can construct a serial schedule S'_i which orders conflicting operations as $<_{S_i}$ and serialises the transactions in T_i as they are ordered in $<_{S_{i+1}}$
Point (1) is true since:
$o <_{S_i} o' \Rightarrow o \rightarrow_{S_{i-1}} o' \Rightarrow \neg(o' \hookrightarrow_{S_{i-1}} o)$, *as otherwise there would be a cycle in* $(\hookrightarrow_{S_{i-1}} \cup \rightarrow_{S_{i-1}})$. *Thus, the transactions of* T_{i-1} *are serialised as* $<_{S_i}$.
Point (2) is true since:
S_i *is CC*
$\Rightarrow$ $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ *is acyclic*
$\Rightarrow$ $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ *can be completed to an acyclic total order which defines the order of the transactions of a serial schedule S'_i which orders conflicting operations as $<_{S_i}$.*
As (1) inductively implies (2), what remains is to show the induction hypothesis. There is a serial schedule S'_1 which orders conflicting operations as $<_{S_1}$ since all its operations are executed sequentially. Therefore, (2) holds without (1) for $i = 1$. □

Theorem 3. *An execution in a fork schedule is correct iff it is FCC.*
Proof. *For every fork schedule FS a semantically equivalent stack schedule SS consisting of SS_1 and SS_2 can be constructed with:*
$SS_2 := S_F$; $T_{SS_1} := O_{S_F}$, $O_{SS_1} := \bigcup_{i=1}^{n} O_{S_i}$, $\rightarrow_{SS_1} := \bigcup_{i=1}^{n} \rightarrow_{S_i}$, $\mapsto_{SS_1} := \bigcup_{i=1}^{n} \mapsto_{S_i}$,
$CON_{SS_1} := \bigcup_{i=1}^{n} CON_{S_i}$ *(i.e.,* $\forall o \in t, o' \in t', t \in T_{S_i}, t' \in T_{S_j}, i \neq j : \neg CON_{SS_1}(o, o')$),
$<_{SS_1} := \bigcup_{i=1}^{n} <_{S_i}$, $\ll_{SS_1} := \bigcup_{i=1}^{n} \ll_{S_i}$.
Now, SS_2 is CC iff S_F is, and SS_1 is CC iff $\bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ is acyclic. Because the fork is FCC then the equivalent stack is SCC. □

The locally testable criterion for FCC can be proven by dividing $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ into subgraphs of the different schedules and considering their connections to each other:

Theorem 4 (Criterion for Fork Conflict Consistency). *A fork schedule FS is fork conflict consistent, iff each of the schedules $S_F, S_1, \ldots, S_n$ is conflict consistent.*

Proof. (Only if). *From FCC follows that $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ is acyclic $\forall i \in \{1, \ldots, n\}$. As all those partial graphs $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ are pairwise unconnected and therefore must also be acyclic, each one of the S_i schedules is CC.*

(If). *If each S_i is CC, all partial graphs $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ $(i \in \{1, \ldots, n\})$ are acyclic. Since there are no conflicts between operations in different S_i, no serialisation graph edge connects different S_i. Hence, all these subgraphs are unconnected and the union of all of all those graphs remains acyclic and, thus, FS is FCC.* □

Theorem 5. *An execution in a join schedule is correct iff it is JCC.*

Proof. (If). *Let $:<:_{JS}$ be the transitive closure of $\lesssim_{JS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$. Then, for every JCC join schedule JS a semantically equivalent stack schedule SS consisting of SS_1 and SS_2 can be constructed with:*

$SS_1 := S_J$; $O_{SS_2} := T_{S_J}$, $T_{SS_2} := \bigcup_{i=1}^{n} T_{S_i}$, $\mapsto_{SS_2} := \bigcup_{i=1}^{n} \mapsto_{S_i}$, $\ll_{SS_2} := \bigcup_{i=1}^{n} \ll_{S_i}$

$CON_{SS_2} := \bigcup_{i=1}^{n} CON_{S_i} \cup \{(o, o') \mid o \in O_t, o' \in O_{t'}, t \in T_{S_i}, t' \in T_{S_j}, i \neq j\}$,

$\rightarrow_{SS_2} := \bigcup_{i=1}^{n} \rightarrow_{S_i} \cup \{(t, t') \mid (\exists o \in O_t, \exists o \in O_{t'} : CON_{SS_2}(o, o')) \wedge \neg(t' :<:_{JS} t)$, such that $\rightarrow_{SS_2}$ is acyclic$\}$.

Clearly, it is possible to construct an acyclic total order $\rightarrow_{SS_2}$ as it is not contradicting $:<:_{JS}$, which is acyclic, and every acyclic partial order can be completed to an acyclic total order. Note that SS_2 has an output order that directly follows from the definition of the new conflicts and its input order. As $\hookrightarrow_{SS_2}$ cannot contradict $\rightarrow_{SS_2}$, $(\hookrightarrow_{SS_2} \cup \rightarrow_{SS_2})$ is also acyclic and from SS_1 is CC follows that SS is SCC.

(Only if). *For a join schedule JS a semantically equivalent stack schedule SS consisting of SS_1 and SS_2 is constructed with:*

$SS_1 := S_J$; $O_{SS_2} := T_{S_J}$, $T_{SS_2} := \bigcup_{i=1}^{n} T_{S_i}$, $\mapsto_{SS_2} := \bigcup_{i=1}^{n} \mapsto_{S_i}$, $\rightarrow_{SS_2} := \bigcup_{i=1}^{n} \rightarrow_{S_i}$,

$CON_{SS_2} := \bigcup_{i=1}^{n} CON_{S_i} \cup \{(o, o') \mid o \in O_t, o' \in O_{t'}, t \in T_{S_i}, t' \in T_{S_j}, i \neq j\}$.

Be SS SCC and have $\lesssim_{JS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ a cycle.

Case (1): From $T \lesssim_{JS} T'$ follows:

$\exists t \in O_T, \exists t' \in O_{T'} : t \hookrightarrow_{SS_1} t', CON_{SS_2}(t, t')$

$\Rightarrow t <_{SS_2} t', t \rightarrow_{SS_1} t'$, as otherwise SS_1 not CC

$\Rightarrow T \hookrightarrow_{SS_2} T'$.

Case (2): From $T \rightarrow_{S_i} T'$ follows: $T \rightarrow_{SS_2} T'$.

Case (3): From $T \hookrightarrow_{S_i} T'$ follows: $T \hookrightarrow_{SS_2} T'$.

Thus, there is a cycle in $(\hookrightarrow_{SS_2} \cup \rightarrow_{SS_2})$ which contradicts SCC.

Note that in both directions CC of SS_1 is equivalent to CC of S_J. □

Theorem 6 (Criterion for JCC). *A completed join schedule CJS is JCC, if each of the schedules $S_J, S_1, \ldots, S_n$ is conflict consistent.*

Proof. *Be $G := \lesssim_{JS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$.*

If all schedules are CC, all subgraphs $G|_{S_i}$ (= restriction of G on nodes of S_i)

are acyclic as $\lesssim_{JS}$-edges exist only between subgraphs. Furthermore, every edge between subgraphs is a $\lesssim_{JS}$-edge.
Assume now a cycle in G. Then, because of the above reasons, there must be a cycle between subgraphs, in general:
$G|_{S_1} \lesssim_{JS} \ldots \lesssim_{JS} G|_{S_n} \lesssim_{JS} G|_{S_1}$.
Then there exist transactions in T_{S_J} with:
$t'_1 \hookrightarrow t_2,\ t'_2 \hookrightarrow t_3,\ \ldots,\ t'_{n-1} \hookrightarrow t_n,\ t'_n \hookrightarrow t_1$.
From definition 14 and CC follows that there exist weak input orders $t'_i \rightarrow t_{i+1}$ and also
$t_i \rightarrow t_{i+1},\ t_i \rightarrow t'_{i+1},\ t'_i \rightarrow t'_{i+1}$.
I.e., there is a cycle $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_1$, which is a contradiction to CC of S_J. □

Theorem 7. *An execution in an FDBS-schedule is correct iff it is FDCC.*
Proof. (If). *Let $:<:_{JS}$ be the transitive closure of $\lesssim_{FDS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$. Then, for every FDCC join schedule FDS a semantically equivalent stack schedule SS consisting of SS_1 and SS_2 can be constructed, whereby SS_1 looks like:*
$T_{SS_1} := \bigcup_{i=1}^{n} T_{S_{Ji}},\ O_{SS_1} := \{o \in O_t \mid t \in T_{SS_1}\},\ \mapsto_{SS_1} := \bigcup_{i=1}^{n} \mapsto_{S_{Ji}}$,
$CON_{SS_1} := \bigcup_{i=1}^{n} CON_{S_{Ji}}$.
Then SS_2 can be defined as:
$O_{SS_2} := \bigcup_{i=1}^{n} T_{S_{Ji}},\ T_{SS_2} := \{t \mid \exists o \in O_{SS_2} : o \in O_t\},\ \mapsto_{SS_2} := \mapsto_{S_F}$,
$CON_{SS_2} := CON_{S_F} \cup \{(o, o') \mid o \in O_t, o' \in O_{t'}, t \in T_S, t' \in T_{S'}, S, S' \in \{S_F, S_{L1}, \ldots, S_{Ll}\}, S \neq S'\}$,
$\rightarrow_{SS_2} := \rightarrow_{S_F} \cup \{(t, t') \mid (\exists o \in O_t, \exists o \in O_{t'} : CON_{SS_2}(o, o')) \wedge \neg(t' :<:_{FDS} t)$, *such that $\rightarrow_{SS_2}$ is acyclic*$\}$.
Clearly, it is possible to construct an acyclic total order $\rightarrow_{SS_2}$ as it is not contradicting $:<:_{FDS}$, which is acyclic, and every acyclic partial order can be completed to an acyclic total order. Note that SS_2 has an output order that directly follows from the definition of the new conflicts and its input order. As $\hookrightarrow_{SS_2}$ cannot contradict $\rightarrow_{SS_2}$, $(\hookrightarrow_{SS_2} \cup \rightarrow_{SS_2})$ is also acyclic and from SS_1 is CC follows that SS is SCC.
(Only if). *For an FDBS-schedule FDS a semantically equivalent stack schedule SS consisting of SS_1 and SS_2 is constructed with SS_1 looking like:*
$T_{SS_1} := \bigcup_{i=1}^{n} T_{S_{Ji}},\ O_{SS_1} := \{o \in O_t \mid t \in T_{SS_1}\},\ \mapsto_{SS_1} := \bigcup_{i=1}^{n} \mapsto_{S_{Ji}}$,
$CON_{SS_1} := \bigcup_{i=1}^{n} CON_{S_{Ji}}$.
Then SS_2 can be defined as:
$O_{SS_2} := \bigcup_{i=1}^{n} T_{S_{Ji}},\ T_{SS_2} := \{t \mid \exists o \in O_{SS_2} : o \in O_t\},\ \mapsto_{SS_2} := \mapsto_{S_F},\ \rightarrow_{SS_2} := \rightarrow_{S_F}$,
$CON_{SS_2} := CON_{S_F} \cup \{(o, o') \mid o \in O_t, o' \in O_{t'}, t \in T_S, t' \in T_{S'}, S, S' \in \{S_F, S_{L1}, \ldots, S_{Ll}\}, S \neq S'\}$.
Be SS SCC and have $\lesssim_{FDS} \cup (\hookrightarrow_{S_F} \cup \rightarrow_{S_F})$ a cycle.
Case (1): From $T \lesssim_{FDS} T'$ follows:
$\exists t \in O_T, \exists t' \in O_{T'} : t \hookrightarrow_{SS_1} t', CON_{SS_2}(t, t')$
$\Rightarrow t <_{SS_2} t', t \rightarrow_{SS_1} t'$, *as otherwise SS_1 not CC*
$\Rightarrow T \hookrightarrow_{SS_2} T'$.

Case (2): From $T \rightarrow_{S_F} T'$ *follows:* $T \rightarrow_{SS_2} T'$.
Case (3): From $T \hookrightarrow_{S_F} T'$ *follows:* $T \hookrightarrow_{SS_2} T'$.
Thus, there is a cycle in $(\hookrightarrow_{SS_2} \cup \rightarrow_{SS_2})$ *which contradicts SCC.*
Note that in both directions CC of SS_1 *is equivalent to CC of* $S_{J1} \ldots S_{Jn}$, *as every* S_{J1} *is CC and there is neither an input order between* $S_{J1} \ldots S_{Jn}$ *nor an output order because there are no conflicts.* □

Theorem 8 (Criterion for FDCC). *A completed FDBS-schedule CFDS is FDCC, if*
(1) each of the schedules $S_F, S_{L1} \ldots S_{Ll}, S_{J1} \ldots S_{Jn}$ *is conflict consistent and*
(2) if $\nexists t, t' \in O_T, T \in T_{S_F} : (t, t' \in T_{S_{Jj}} (j \in \{1 \ldots n\}, \exists t_{L1} \ldots t_{Lk} \in \bigcup_{i=1}^{l} O_{S_{Li}} :$ $(t \hookrightarrow_{S_{Jj}} t_{L1} \hookrightarrow_{S_{Jj}} \ldots \hookrightarrow_{S_{Jj}} t_{Lk} \hookrightarrow_{S_{Jj}} t'))$.
Proof. *Be* $G := \lesssim_{FDS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$.
If all schedules are CC, all subgraphs $G|_{S_i}$ *are acyclic as* $\lesssim_{FDS}$*-edges exist only between subgraphs. Furthermore, every edge between subgraphs is a* $\lesssim_{FDS}$*-edge. Assume now a cycle in G. Then, because of the above reasons, there must be a cycle between subgraphs, in general (let* $:<:_S := (\hookrightarrow_S \cup \rightarrow_S)$*):*
$T_{F11} :<:_{S_F} \ \ldots \ :<:_{S_F} T_{F1j_1} \lesssim_{FDS} T_{L11} \lesssim_{FDS} \ \ldots \ \lesssim_{FDS} T_{L1k_1} \lesssim_{FDS} T_{F21}$
$:<:_{S_F} \ \ldots \ :<:_{S_F} T_{F2j_2} \lesssim_{FDS} T_{L21} \lesssim_{FDS} \ \ldots \ \lesssim_{FDS} T_{Lmk_m} \lesssim_{FDS} T_{F11}$.
We know that all pairs of transactions in this cycle are different from each other, as otherwise the join schedule at which their subtransactions execute would detect a contradiction to condition (2) of this criterion.
Then there exist transactions $t_{F11} \in T_{F11}, \ldots$ *with:*
$(t_{F1j_1} \hookrightarrow_{S_{J1}} t_{L11} \hookrightarrow_{S_{J1}} \ \ldots \ \hookrightarrow_{S_{J1}} t_{L1k_1} \hookrightarrow_{S_{J1}} t_{F21}), \ldots,$
$(t_{Fm,j_m} \hookrightarrow_{S_{Jn}} t_{Lm1} \hookrightarrow_{S_{Jn}} \ \ldots \ \hookrightarrow_{S_{Jn}} t_{Lmk_m} \hookrightarrow_{S_{Jn}} t_{F11})$
There exist (a.o.) the following conflicts in S_F*:*
$(t_{F1j_1}, t_{F21}), \ldots, (t_{Fm-1,j_{m-1}}, t_{Fm1}), (t_{Fmj_m}, t_{F11})$.
Conflicting operations are weakly output ordered, and as they are transactions in one S_Jj, *resp., also weakly input ordered.*
$\Rightarrow T_{F11} :<:_{S_F} \ \ldots \ :<:_{S_F} T_{F1j_1} \hookrightarrow_{S_F} T_{F21} \ldots$.
This is a cycle in $(\hookrightarrow_{S_F} \cup \rightarrow_{S_F})$, *hence, a contradiction to CC of* S_F. *Thus, G must be acyclic and CFDS is FDCC.* □