Correctness in General Configurations of Transactional Components

Gustavo Alonso, Armin Feßler, Guy Pardon*and Hans-Jörg Schek
Institute of Information Systems, Swiss Federal Institute of Technology (ETH)
ETH Zentrum, CH-8092 Zürich, Switzerland
{alonso,fessler,pardon,schek}@inf.ethz.ch

Abstract

From a transactional point of view, composite systems are component based applications in which each component has its own transaction management logic. These systems are highly relevant in practice since they are likely to be the standard architecture for many future distributed applications. Unfortunately, there is no appropriate conceptual framework in which to reason about such systems. Following up on existing work that addressed special cases of composite systems, in this paper we tackle the problem of general composite systems, i.e., those with arbitrary configurations. We propose a correctness criterion, develop a new proof technique that allows us to address arbitrary configurations, and discuss several important issues related to concurrency control in distributed systems.

1 Introduction

Many of today's distributed systems follow a multiple component approach where the components are structured in successive invocation layers. In such systems, when a component invokes another component, it explicitly delegates part of the execution to the called party, which, in turn, may call upon the services of yet other components. In this paper, we address the problem of concurrent invocations when each of these components has its own scheduler, its own transactions, and can execute operations that are again transactions on some other component. The resulting system is what we call a *composite system*. Such complex transactional interactions require careful analysis in order to determine what is a correct execution.

However, to date, there is no solid general theoretical framework in which to reason about this type of systems. In particular, most work done in this area either assumes a given configuration or introduces impractical assumptions. For instance, in multilevel transactions [Wei91], the system is always configured as a sequence of schedulers where the output of one constitutes the input to the next. This configuration is what we call a *stack* and we have shown that special cases of the composite model already provide correctness classes that are larger than those based on multilevel transactions [ABFS97]. Within multilevel transactions, level-by-level serializability (LLSR) is a good example of the type of restrictions we would like to avoid. In order to allow independent schedulers at each level, LLSR assumes that if two operations conflict at one level, they must also conflict at all lower levels. While conceptually this is an elegant solution, in practice this means that the design of each level has to be done taking into consideration all other levels in the system, eliminating the modularity which we see as the main appeal of composite systems. As an alternative, the notion of *order preserving serializability* (OPSR) also allows independent scheduling in similar configurations[BBG89]. As shown in [ABFS97], OPSR is also a special case of the composite model.

Another model to be considered for composite systems is nested transactions [Mos85] which, at least in theory, does not restrict the configuration of the system. However, and to our knowledge, all existing implementations of nested transactions are centralized (in fact, nested transactions are more a concurrency control mechanism than a correctness criterion). Moreover, nested transactions assume that all transactions share at least one scheduler and can therefore be related to one another. This premise does not hold in composite systems, where two transactions may not have any scheduler in common and still interfere with each other through transitive dependencies.

As a first step towards developing a theory of composite systems, in some of our previous work we have studied special cases of composite executions (stack [ABFS97], fork and join [AFPS99]) in an isolated context. In this paper, we generalize these ideas and provide the necessary concepts and techniques to be able to discuss general composite systems. The challenge behind the general case is the arbitrary nature of the configuration. For simple configurations, e.g. stack and fork compositions [AFPS99], relatively simple correctness criteria can be derived. For the general case, these approaches do not work and a more complex theory needs to be developed. To define what is a correct execution in a general composite system, we borrow some of the proof techniques developed by Beeri et al. [BBG89] although modified to adapt them to the special characteristics of composite systems. Our approach to define correctness consists of several steps. First, we sort the schedules in the system to determine the sequence to consider for correctness purposes (i.e., whether a schedule is "below", "above", or "at the same level" as another schedule). Second, we introduce a gener-

*Part of this work has been funded by ETH Zürich within the DRAGON project(Reg-Nr 41-2642.5)

alized conflict relation in order to capture conflicts from a lower level that must be considered at a higher level, where they give rise to an observed order. In a way, conflicts are (conceptually) pulled up, like in [BBG89, Wei91, SWS91], as opposed to our work in [AFPS99] where we pushed down order constraints along the invocation hierarchies. Based on the schedule sequence and on the observed order, we try to reduce the composite system to an execution over the roots. If we succeed, we consider the composite system to be correct, otherwise, the system is deemed to be incorrect. Later on, we prove that this approach is equivalent to disentangling the execution trees in the composite system, i.e., akin to finding a serializable execution of the composite transactions.

Our contribution lies in developing a formal framework for general composite systems, along with a technique for proving arbitrary composite systems correct. In addition, we provide a deeper understanding of the concurrency control problems that arise in such systems and show how a variety of interesting configurations can be seen as special cases of the general scenario. We expect these ideas to be of practical use as the component-based approach gains popularity in applications such as CORBA-based systems, TP-monitors, or Web-based information systems. In terms of implementation strategies, we are considering combinations of open nested transactions [BSW88, Sch96] and closed nested transactions [Mos85, GR93]. These strategies lead to different restrictions and we are currently looking into them to find appropriate concurrency control protocols with which to implement general composite systems. For reasons of space, in this paper we concentrate solely on the formal aspects.

The rest of the paper is organized as follows. Section 2 defines composite systems. Section 3 discusses composite correctness. Finally, Section 4 compares the resulting correctness notion to existing notions.

2 Composite Systems

In this paper, we will use the term *schedule* (defined more precisely below), to refer to the result of the dynamic behaviour of a *scheduler*. A general composite system consists of a set of schedulers that invoke each other's services (Figure 1). In such system, a composite transaction can be seen as a *tree*. The entire composite schedule is the result of the combination of the individual schedules produced by each scheduler or component in the system. In the composite schedule, the dynamic behaviour of the system as a whole is depicted with execution trees, or, more formally, with a *computational forest* [BBG89]. With the exception of recursion (a schedule directly or indirectly invoking itself), we allow any arbitrary configuration.[1] In a computational forest, not all transactions have the same height and it is possible for some transactions not to have any schedules in common (e.g., T_1 and T_6 in Figure 1).

As a first step to understand arbitrary composite systems, we need to formally define how the schedules interact with each other, what is a composite transaction, and how each schedule should behave. The model we use for this purpose is based on the notions of *weak* and *strong* orders [ABFS97]:

[1] Recursion can be dealt with by applying certain transformations to the configuration. These, however, are not too interesting and complicate the exposition a great deal. In what follows, we will simply exclude recursion to simplify the presentation.

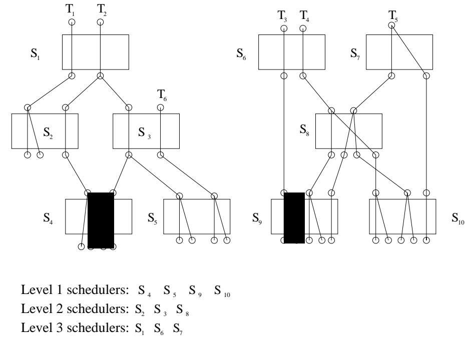


Figure 1: Examples of composite systems

Definition 1 (Strong and Weak Order:) *Let A and B denote two transactions. There are three possible order relations between them:*

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

These orders are, in all cases, transitively closed. The reason to distinguish between these orders is that they allow a much more accurate description of scheduling requirements. Current models do not distinguish between these orders and, as a result, offer only restricted degrees of parallelism. Moreover, these orders allow us to define how schedules interchange information but this requires to slightly redefine traditional concepts, leading to a new transaction model which we termed *composite transactions.* Thus, a transaction is now defined as follows. Let $\widehat{O}$ be the set of all operations from which transactions can be formed (note that elements of $\widehat{O}$ can again be transactions).

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* $\ll_t \subseteq <_t$. □

The weak and strong intra-transaction orders are requirements imposed by each transaction regarding how its operations need to be executed. Strong order corresponds to operations that need to be executed in strict temporal order (one after the other) and the weak order specifies in which direction data flows should occur (always in the direction of the weak orders). Furthermore, two operations are considered to conflict if they do not commute (i.e., if their relative order of execution matters). From here, we can define a schedule as follows:

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

- *T is a set of transactions, with O denoting the set of all operations in T, i.e.,* $O = \bigcup_{t \in T} O_t$.

- *$CON_S \subseteq O \times O$ is a conflict predicate.*
- *$\rightarrow$ 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. *$\forall t, t' \in T, t \neq t', \forall o \in O_t, \forall o' \in O_{t'}, CON_S(o, o')$:*
     (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. *$\ll \subseteq <$.* □

This notion of schedule abstracts each element of a composite system and defines what inputs it should receive and what outputs to produce. The important thing to notice in this definition is that weak orders are only propagated (from input to output level) when operations conflict, otherwise the weak order disappears. This allows to increase parallelism when a schedule can use semantic knowledge to ascertain that two operations do not conflict although their respective transactions are weakly ordered.
To generalize this idea, we consider a composite system to be a collection of schedules. Let T_S be the set of transactions at schedule S, O_S the set of operations of schedule S and CON_S the conflict relationship defined for O_S. $\rightarrow_S$ is the weak input order and $<_S$ the weak output order of S, respectively. Note that the weak order includes the strong order and thus the latter does not have to be explicitly considered.

Definition 4 (Composite System) *A* composite system CS *consists of n schedules $S_1, \ldots, S_n$, such that:*

1. *$\forall S_i, S_j \in CS, i \neq j : T_{S_i} \cap T_{S_j} = \emptyset$.*
2. *$\forall S_i \in CS$:*
   a) if $t \in O_{S_i} : t \in T_{S_j}$ then S_i is called a leaf schedule.
   b) if $\exists t \in O_{S_i} : t \in T_{S_j}$ then S_i is called an internal schedule.
3. *A leaf operation (or leaf) is every element of the set L: $\{o | o \in O_{S_i} \wedge S_j : o \in T_{S_j}\}$.*
4. *An internal (transaction) node is every element of the set I: $\{t | t \in O_{S_i} \wedge \exists S_j : t \in T_{S_j}\}$.*
5. *A root transaction is every element of the set R, defined as: $\{r | S : r \in O_S\}$.*
6. *Let the transitive closure of the set of operations of some transaction T be denoted by $Act(T)$ then: $\forall S_i, S_j \in CS, i \neq j : \forall (T_1, T_1') \in (T_{S_i} \times T_{S_j}), T_1' \in Act(T_1)$ it holds: $(T_2, T_2') \in (T_{S_i} \times T_{S_j}) : T_2 \in Act(T_2')$. Additionally, we impose: $\forall S_i \in CS, T \in T_{S_i}, T' \in Act(T) : T' \notin T_{S_i}$*
7. *$\forall t, t' \in O_{S_i}$ with $t, t' \in T_{S_j}$:*

$$\begin{cases} t <_{S_i} t' & \Rightarrow & t \rightarrow_{S_j} t' \\ t \ll_{S_i} t' & \Rightarrow & t \mapsto_{S_j} t' \end{cases}$$

The first point specifies that every transaction is assigned to a single schedule. The second point states that a transaction's operations can in turn be transactions of some other schedule. Note that an internal schedule can also have leaf operations. The elements of the sets L, I, R in points 3 to 5 will be called *nodes*, because they represent transactional nodes in the execution trees of the system. Point 6 introduces a limitation on subtransactions and the schedules' invocation hierarchy: none of the schedules (directly or indirectly) invoked by S are in turn clients of S (this effectively prevents recursion, i.e., a schedule which directly or indirectly invokes itself). Finally, the last point requires output orders from a schedule to be passed on as input orders to another schedule if both operations are sent to the latter.

Definition 5 (Parent) *The* parent *of an operation or transaction t is defined as follows:*

- *if $\exists T : t \in O_T$ then $parent(t) = T$.*
- *otherwise (if t is a root transaction), $parent(t) = t$.* □

Definition 6 (Composite Transaction (CT))
A composite transaction *is a set of transactions/operations $CT = \{T_1, T_2, ..., T_n\}$ such that:*

1. *$T_1 \in R$*
2. *$\forall T_i, i \in [2..n] : T_i \in Act(T_1)$*
3. *$T_j : \neg(T_j \in CT) \wedge T_j \in Act(T)$* □

Thus, a composite transaction is a root transaction and all its *descendants*. From now on, we will also use 'execution tree' as an equivalent term for composite transaction.

3 Correctness of a Composite System

Our discussion of correctness will be based on a bottom-up analysis of a given execution. This is due to the potential interference between subtransactions which has implications on the parents, especially if those parents are in different schedules that do not know about each other. This approach does not preclude practical protocols from working top-down, i.e., by enforcing restrictions on how the subtransactions can be executed. An example of such protocol is CC scheduling [ABFS97, AFPS99].

3.1 Structure of a Composite System

Although a composite system can be a very complex interconnection of schedules, the absence of recursive client-server relationships allows us to introduce some structure. This is critical for the reduction process later on, since it allows us to perform a stepwise abstraction of the composite schedule independently of its configuration. To define this structure, we need some initial formalism:

Definition 7 (Invokes (Inv)) *A schedule S_i* invokes *schedule S_j (S_i Inv S_j) if $\exists t \in O_{S_i} : t \in T_{S_j}$.* □

Definition 8 (Invocation Graph (IG))
An invocation graph is defined as follows:

1. *The nodes of the graph are the different schedules in the system.*
2. *Insert a directed edge e_{ij} from S_i to S_j if S_i Inv S_j.* □

Since a composite system does not allow recursion, the IG is acyclic.

Definition 9 (Level of a Schedule)
For a schedule S, its level is determined as follows:

- *let l be the length of the longest path that starts at S in the IG.*
- *the level of S is defined as $l+1$.* □

Because IG has no cycles, the level of a schedule is finite and well-defined.
A transaction/operation that belongs to a schedule whose level is i will from now on be called a *transaction/operation of level i*. Moreover, a composite system is said to be of order N, if N is the highest level of any schedule in the system. See Figure 1 for an example of how the elements are numbered.

3.2 Structure of Composite Histories

In a composite system, it is possible for transactions not to share any schedule (for instance, transactions T_1 and T_6 in Figure 1). It is necessary, however, to be able to establish relations among transactions in the composite system. We do this with the notion of observed order, which propagates upwards the dependencies among transactions:

Definition 10 (Observed order) *The observed order $<_e$ between operations t, t' (either leaves or internal nodes) is defined as follows:*

1. *If either t or t' is a leaf on schedule S_i, then $t <_e t'$ iff $t <_{S_i} t'$.*
2. *If $t <_{S_i} t' \wedge CON_{S_i}(t, t')$ then $parent(t) <_e parent(t')$.*
3. *If $t <_e t' \wedge \neg(t, t' \in O_{S_i})$ then also $parent(t) <_e parent(t')$.*
4. *If $t <_e t' \wedge t' <_e t''$ then also $t <_e t''$ (transitivity rule).* □

Point 1 establishes the atomicity of leaf operations. Point 2 says that when two operations conflict, the observed order is introduced among their parents (similar to the notion of *quasi-order* used by [Wei91]; in classical concurrency theory, this is the serialization order). Propagation of the observed order occurs as follows. As long as the ancestors of the parents do not go through a common schedule, the observed order is pushed up further (point 3; an extension of the *ghost-graph* in [AFPS99]). When a common schedule is found, the observed order is not propagated if there is no conflict at the common schedule (each schedule is responsible for conflicts between its own operations – if the operations in a schedule do not conflict then this schedule 'knows' that there is commutativity and this knowledge can be exploited for reasoning about correctness). Note that $<_e$ may not be acyclic, just like the classical notion of serialization order does not guarantee acyclicity.
The observed order allows us to obviate the different schedules and relate transactions independently of the configuration. However, we still have the problem that conflicts are only defined within schedules. To generalize the execution trees entirely, we also need to generalize the conflict relation as follows:

Definition 11 (Generalized Conflict Relationship)
Two operations t, t' (elementary or subtransactions) are said to conflict, *$CON(o, o')$, if one of the following holds:*

1. *$o, o' \in O_{S_i} \wedge (o, o') \in CON_{S_i}$*
2. *$\neg(o \in O_{S_i} \wedge o' \in O_{S_i}) \wedge o <_e o'$.* □

The generalized conflict relationship (an extension to the dynamic conflict relation in [SWS91]) merely states that for operations in the same schedule, the conflict relation is that of the schedule. For operations in different schedules, we will assume a conflict if they are related by $<_e$ (i.e., if there has been interaction at a lower level). From now on, we will use CON to refer to the generalized conflict relationship. As an example of these ideas, consider Figure 2. At the leaf level, there are o_{13}, o_{23} on S_3, both conflicting and ordered by S_3. Therefore, they are also in the observed order and have a generalized conflict relationship. Going up, we can incrementally relate (T_1, T_2) and (T_1, T_4) both according to the conflict and the observed order definition. Note that we do not relate every possible pair of transactions, only those for which an observed order/conflict relation exists. Conflicts can also disappear. For instance, consider Figure 3 (f). Although a conflict exists between t_{211} and t_{121}, this conflict is no longer relevant in the next phase shown in Figure 3 (g). The reason is that the respective parents are scheduled by the same schedule, which knows about the semantics of its operations and guarantees there is no conflict among them.

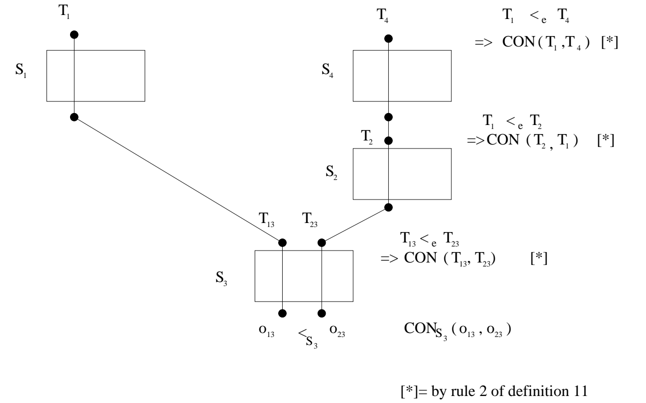


Figure 2: Illustration of conflict and observed order

3.3 Handling Composite Executions

Definition 12 (Computational Front) *A computational front is a quadruple $F = (O, \rightarrow, <_e, CON)$ where O is a maximal set of independent nodes (i.e., a set of nodes of which none is a descendant of any other), $<_e$ and CON are defined as explained earlier and constructed for all nodes in F. Finally, $\rightarrow$ is the set of all input orders between elements of O.* □

This definition is based on an idea suggested in [BBG89]. Because $\mapsto \subseteq \rightarrow$, a front implicitly takes strong input orders into account. A front represents any independent set of operations that will accomplish the job of a root transaction. At the lowest level, this means the entire set of leaves. At the highest abstraction level, this means the abstract execution of the roots.

Definition 13 (Conflict Consistency (CC)) *A front F is conflict consistent iff the union of its* $<_e$ *and* $\rightarrow$ *is acyclic.* □

In [AFPS99], CC is defined in a slightly different way, but it is also shown that the property expressed by definition 13 is equivalent to CC.

3.4 Reducing a Composite System

Definition 14 (Calculation of a transaction) *A calculation of a transaction* $T = (O_t, <_t)$ *in a front F is defined as an isolated execution sequence in F, involving all operations of T, in which the observed order* $<_e$ *does not contradict* $<_t$*. By isolated execution sequence of T, we mean one in which all operations of T appear together, with no other operations* $<_e$*-ordered in between them.* □

The previous definition is important for simplifying an execution by reduction. If we can construct a calculation of transaction T, then we can eliminate the corresponding lower-level operations and use T instead. A calculation is similar to the idea of an isolated tree in nested transactions.

Definition 15 (Level 0 Front) *A level 0 front is a front F for which:*
O is the set of all leaves. □

By definition, for any composite schedule, there is always a level 0 front (and only one).
Starting from the level 0 front, we now perform stepwise reduction, in a level-by-level way. This means that in each step i, the operations of all schedules of level i will be reduced and thus replaced by their transactions.

Definition 16 (Level i Front) *A level i front F is constructed from a level (i-1) front F* through the following steps (in this order):*

1. *Construct another (i-1) front F** out of F* (by changing the order of commuting pairs of operations – without switching operation pairs that are ordered by* $\rightarrow$ *), in which each level i transaction* T_i *is represented by a calculation.*
2. *Replace the operations* O_{T_i} *of each* T_i *by* T_i *(reduction step).*
3. *Include the relevant* $<_e$ *and CON pairs involving* T_i *as specified in the definitions.*
4. *Delete the* $<_e$ *and CON pairs involving any* $o_i \in O_{T_i}$*.*
5. *If* $\exists T_i \in F* : parent(T_i) = T_i$ *then* $T_i \in F$ *(propagation step).*
6. *Include the relevant* $\rightarrow$ *pairs of each level i schedule, and check that the resulting front is CC.* □

Point 5 explicitly states that root transactions are kept in the next level front, because we want to arrive at a front only including all root transactions. Note that a level i front does not necessarily exist, since it may not be possible to find F** in step 1 or because it is not CC (step 6). Intuitively, a level i front (if it exists) is equivalent to the level (i-1) front, but simpler because less operations appear in it. Nevertheless, because of the definitions of $<_e$ and CON, all relevant information is kept. It is not difficult to see that in the new front there is a conflict between two operations a, b whenever one of the following holds:

- a, b are both operations of the same schedule S_i and this schedule 'says' that there is a conflict.
- a, b are not operations of the same schedule but are related by $<_e$, in which case it is not known whether they really conflict so a conflict is assumed pessimistically (because $<_e$ originates from accesses to common data items on a lower level).

From here, we can define correctness in a composite system.

3.5 Correctness in a Composite System

As in conventional theory, we consider a serial front to be correct by definition.

Definition 17 (Serial Front) *A front F is serial if* $\mapsto$ *is a total strong order, i.e.,* $\forall t, t' \in F : (t \mapsto t') \vee (t' \mapsto t)$*.* □

We will define correctness in terms of equivalence to a serial front. However, given the structure of a composite system, this cannot be done directly since fronts do not necessarily imply the same configuration. To resolve this, we use:

Definition 18 (Level-i-Equivalence) *A CS is level-i-equivalent to a front F, iff CS has a level i front that is identical to F.*

This definition allows composite systems to be compared, even without having the same structure since the front F can be some level j front of another CS, with $i \neq j$. In that case, what happens on lower levels is irrelevant, as long as the effect on the levels i and j is the same. From here we define:

Definition 19 (Level-i-Containment) *A CS is level-i-contained in a front F, iff*

1. *CS is level-i-equivalent to a front F*,*
2. $\rightarrow_F$ *contains* $\rightarrow_{F^*} \cup <_{e_{F^*}}$*, and*
3. $O_F = O_{F^*}, CON_F = CON_{F^*}$*.*

Definition 20 (Composite Correctness (Comp-C)) *A composite schedule CS is correct iff it is level-N-contained in a serial front* F_S*.* □

This definition is the equivalent of the traditional notion of conflict equivalent serializability [BHG87] but adapted to composite systems. With the reduction technique, a more straightforward approach to check for correctness is to use the following theorem:

Theorem 1 *A composite schedule CS is correct iff CS has a level N front.*

That is, if we can complete the reduction process all the way to the roots, then the composite system is Comp-C. Proof of this theorem can be found in the appendix.

3.6 An Incorrect Execution

Figure 3 shows an example of an execution that is not Comp-C. This can be seen by following the reduction process. In the level 1 front, there are two conflicts, introduced by 'pulling' up the information about the conflicts on level 0 (because the transaction pairs involved originated on different schedules). Further reduction is still possible and the level 2 front is shown at the bottom of the left column. However, here it is no longer possible to construct an isolated execution for T_1, implying incorrectness of the initial schedule.

3.7 A Correct Execution

On the other hand, Figure 4 shows a correct execution. A level 1 front is shown, along with pulled-up conflicts and observed order. Going on with the technique, we find a level 2 front. Because t_{21} and t_{12} are both operations of the same schedule and that schedule (the level 3 schedule) does not indicate a conflict, the orders obtained for t_{211} and t_{121} in the previous step are forgotten (since they can be trusted to be irrelevant). The final step leads to a level 3 front with only root transactions, shown in the next figure.

4 Discussion

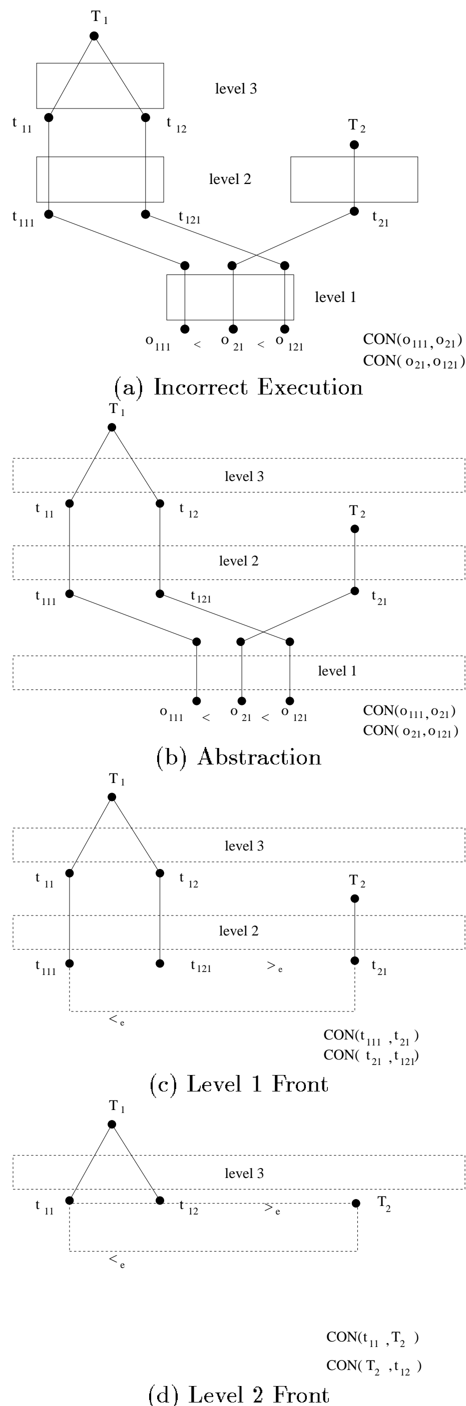
In this paper, we have dealt with the issue of correctness in general recursion-free composite systems. The correctness criterion (Comp-C) as defined in the previous section is more general than previous results such as stack conflict consistency (SCC) [ABFS97], fork conflict consistency (FCC) and join conflict consistency (JCC) [AFPS99]. This should not be surprising, since all these previous architectures are special configurations of the general composite system considered here. Thus:

Theorem 2 *For a stack architecture, SCC* $\Leftrightarrow$ *Comp-C.*

Theorem 3 *For a fork architecture, FCC* $\Leftrightarrow$ *Comp-C.*

Theorem 4 *For a join architecture, JCC* $\Leftrightarrow$ *Comp-C.*

Proof of these theorems can be found in the appendix. What is important here is that as a result of these theorems, we can claim that the composite systems theory presented here is more general and encompassing than existing models. For instance, in [ABFS97], it was shown that level-by-level serializability, multilevel serializability [Wei91], and order preserving serializability [BBG89] are all subsets of SCC. Therefore, they are also proper subsets of Comp-C. Moreover, in [AFPS99] it is shown how the stack, fork and join can be used to model a variety of transaction models like federated transactions, the ticket method for federated transaction management, sagas and distributed transactions. The results in this paper show that Comp-C is a framework where all these models can be understood and compared. We are currently working on using the ideas of this paper to describe a variety of transaction models and on practical applications of the composite theory. We are also in the process of implementing a prototype composite system in which to test these ideas [PA].



(a) Incorrect Execution

(b) Abstraction

(c) Level 1 Front

(d) Level 2 Front

Figure 3: Example of incorrect executions

References

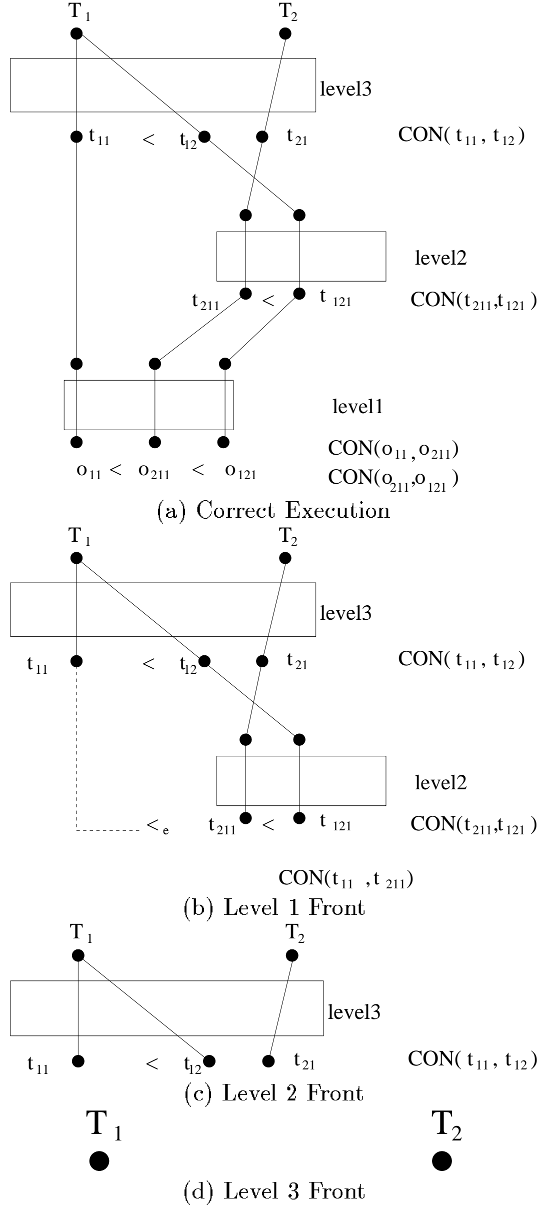
[ABFS97] Gustavo Alonso, Stephen Blott, Armin Fessler, and Hans-Jörg Schek. Correctness and parallelism in composite systems. In *PODS97*, 1997.

[AFPS99] G. Alonso, A. Feßler, G. Pardon, and H.-J. Schek. Transactions in stack, fork and join composite systems. In *Int. Conference on Database Theory*, 1999.

[BBG89] Catriel Beeri, Philip A Bernstein, and Nathan Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BSW88] Catriel Beeri, Hans-Jörg Schek, and Gerhard Weikum. Multi-level transaction management, theoretical art or practical need? In *International Conference on Extending Database Technology*, 1988.



(a) Correct Execution

(b) Level 1 Front

(c) Level 2 Front

(d) Level 3 Front

Figure 4: Example of correct executions

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[PA] Guy Pardon and Gustavo Alonso. Plug and play transactional resource management over the internet. In Preparation.

[Sch96] Werner Schaad. *Transaktionsverwaltung in heterogenen, foederierten Datenbanksystemen.* PhD thesis, ETH Zuerich, 1996.

[SWS91] 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, April 1991.

[Wei91] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM TODS*, 1991.

Appendix: Proofs

Theorem 1 *A composite schedule CS is correct iff CS has a level N front.*

Proof. To show: CS is level-N-contained in a serial front $F_S \Leftrightarrow$ CS has a level N front

(if)

The given level N front $(O, \rightarrow, <_e, CON)$ is, by definition, CC, i.e., $(<_e \cup \rightarrow)$ is acyclic. By topological sorting, we convert $(<_e \cup \rightarrow)$ into a total order $\rightarrow_S$ over O. Then, CS is level-N-contained in $F_S = (O, \rightarrow_S, <_e, CON)$ since:

1. CS is level-i-equivalent to $F^* = (O, \rightarrow, <_e, CON)$
2. $\rightarrow_S$ contains $(<_e \cup \rightarrow)$
3. $O_{F_S} = O, CON_{F_S} = CON$

With $\mapsto_S := \rightarrow_S$, F_S is serial and, thus, CS is correct.

(only if)

Given a correct schedule CS, we know that by definition there is a serial front F_S that level-N-contains CS. This implies that CS is level-N-equivalent to some front F* with: $\rightarrow_{F_S}$ contains $(<_{e_{F^*}} \cup \rightarrow_{F^*})$ and $O_{F_S} = O_{F^*}, CON_{F_S} = CON_{F^*}$. This, in turn, means that CS has a level N front F*. □

In the following, we will use CC both in relation to a front (as defined in this paper) and a schedule (as defined in earlier work). However, from the context it should be clear which interpretation is meant.

Now, we will repeat the relevant definitions concerning stack, fork, and join schedules introduced in [AFPS99] and prove that SCC, FCC, JCC is Comp-C, resp.

Definition 21 (Stack Schedule (SS))
*SS, an n-*level stack schedule*, consists of n schedules $S_1, \ldots, S_n$, such that, for $1 < i \leq n$:*

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

Definition 22 (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 $1 \leq i \leq n$.* □

With this, we can prove that the new correctness criterion Comp-C includes the former one of stack conflict consistency:

Theorem 2 *For a stack architecture, SCC $\Leftrightarrow$ Comp-C.*

Proof. (only if)
It is a trivial task to identify the level of each scheduler in this architecture. We will start with the bottom level scheduler S_1 and work our way up. We must show that an SCC n-level stack schedule SS has a level n front. The level 0 front is $F_0 = (O_{S_1}, <_{S_1}, <_{S_1}, CON_{S_1})$. Since S_1 is CC, there exists a serial schedule S_1^{ser}, such that $F_1 = (O_{S_2}, \rightarrow_{S_1^{ser}}, \hookrightarrow_{S_1^{ser}}, CON_{S_2})$ is a level 1 front of SS, as $\rightarrow_{S_1^{ser}}$ contains $\rightarrow_{S_1}$ and $\hookrightarrow_{S_1^{ser}} = \hookrightarrow_{S_1}$. Because of the CC property of each scheduler in the stack, the same applies to the second level scheduler. Eventually, we reach the top level (n) scheduler. This means that SS has a level n front, implying Comp-C.
(if)
Be SS Comp-C, i.e., it has a level n front and, by definition, level i fronts for $i \in \{1 \dots n-1\}$. From definition 20 and theorem 1 follows that every stack schedule SS_i consisting of $S_1 \dots S_i (1 \leq i \leq n)$ is level-i-contained in a serial front F_i. So, SS_i is level-i-equivalent to front F* with $\rightarrow_{F_i} \supseteq (\rightarrow_{F^*} \cup <_{e_{F^*}})$, which implies $\rightarrow_{F_i} \supseteq (\rightarrow_{S_i} \cup <_{e_{S_i}})$ and consequently, every S_i is CC. Thus, SS is SCC. □
The corresponding correlation also holds for fork conflict consistency:

Definition 23 (Fork Schedule (FS)) *A fork schedule FS consists of (n+1) schedules $S_F, S_1, \dots, S_n$, such that:*

1. $O_{S_F} = \bigcup_{i=1}^{n} T_{S_i}$
2. $\forall i \in \{1, \dots, n\} : \forall t, t' \in T_{S_i}$: $\begin{cases} t <_{S_F} t' \Rightarrow t \rightarrow_{S_i} t' \\ t \ll_{S_F} t' \Rightarrow t \mapsto_{S_i} t' \end{cases}$
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.* □

Definition 24 (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 *For a fork architecture, FCC $\Leftrightarrow$ Comp-C.*

Proof. (only if)
Just as in the stack case, levels are trivially found for each scheduler of a fork schedule FS: $S_1 \dots S_n$ are of level 1 and S_F is of level 2. The FCC criterion implies that each of the schedules involved is conflict consistent. We must show that FS has a level 2 front, and for this to be possible, a level 1 front. The level 0 front is $F_0 = (\bigcup_{i=1}^{n} O_{S_i}, \bigcup_{i=1}^{n} <_{S_i}, \bigcup_{i=1}^{n} <_{S_i}, \bigcup_{i=1}^{n} CON_{S_i})$. Since all S_i are CC, and there are no conflicts between operations of different S_i, there exist serial schedules S_i^{ser}, such that $S_1^{ser} = (\bigcup_{i=1}^{n} T_{S_i}, \bigcup_{i=1}^{n} \rightarrow_{S_i}, \bigcup_{i=1}^{n} \mapsto_{S_i}, \bigcup_{i=1}^{n} <_{S_i}, \bigcup_{i=1}^{n} \ll_{S_i}, \bigcup_{i=1}^{n} CON_{S_i})$. Now, $F_1 = (O_{S_2}, \rightarrow_{S_1^{ser}}, \hookrightarrow_{S_1^{ser}}, CON_{S_2})$ is a level 1 front of SS, as $\rightarrow_{S_1^{ser}}$ contains $\bigcup_{i=1}^{n} \rightarrow_{S_i}$ and $\hookrightarrow_{S_1^{ser}} = \bigcup_{i=1}^{n} \hookrightarrow_{S_i}$. Because of the CC property of S_F, the same argumentation holds with S_2^{ser} being a serial schedule whose weak input order contains that one of S_F. This means that FS has a level n front, implying Comp-C.
(if)
For a fork schedule FS to be Comp-C, it must have a level 2 and level 1 front. From definition 20 and theorem 1, it follows that S_F is level-2-contained in a serial front F_2 and the lower level schedule $S_{low} = (\bigcup_{i=1}^{n} T_{S_i}, \bigcup_{i=1}^{n} \rightarrow_{S_i}, \bigcup_{i=1}^{n} \mapsto_{S_i}, \bigcup_{i=1}^{n} <_{S_i}, \bigcup_{i=1}^{n} \ll_{S_i}, \bigcup_{i=1}^{n} CON_{S_i})$ is level-1-contained in a serial front F_1. So, FS is level-2-equivalent to a front F_2^* with $\rightarrow_{F_2} \supseteq (\rightarrow_{F_2^*} \cup <_{e_{F_2^*}})$, which implies $\rightarrow_{F_2} \supseteq (\rightarrow_{S_F} \cup <_{e_{S_F}})$ and thus, by definition, S_F is CC.
On the other hand, the lower schedule S_{low} is level-1-equivalent to a front F_1^* with $\rightarrow_{F_1} \supseteq (\rightarrow_{F_1^*} \cup <_{e_{F_1^*}})$, which implies $\rightarrow_{F_1} \supseteq (\rightarrow_{S_{low}} \cup <_{e_{S_{low}}})$. By definition, S_{low} is then CC and, since parts of acyclic graphs are also acyclic, all schedules $S_1 \dots S_n$ are CC.
Thus, FS is FCC. □
To prove the correlation JCC $\Leftrightarrow$ Comp-C, we not only need the definition of join conflict consistency (JCC), but also the definition of the ghost-graph, as the former is based on the latter:

Definition 25 (Join Schedule (JS)) *A join schedule JS consists of (n+1) schedules $S_J, S_1, \dots, S_n$, such that:*
- $\bigcup_{i=1}^{n} O_{S_i} = T_{S_J}$
- $\forall i \in \{1, \dots, n\} : \forall t, t' \in O_{S_i}$: $\begin{cases} t <_{S_i} t' \Rightarrow t \rightarrow_{S_J} t' \\ t \ll_{S_i} t' \Rightarrow t \mapsto_{S_J} t' \end{cases}$ □

Definition 26 (Ghost-Graph for Join Schedules($_{JS}$)) *$\forall (T, T')$ with $T \in T_{S_i}, T' \in T_{S_j}, i \neq j$ the* ghost-graph $_{JS}$ *is defined as:*
T $_{JS}$ 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 27 (Join Conflict Consistency (JCC)) *A join schedule JS is* join conflict consistent*, if the schedule S_J is conflict consistent and $_{JS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ is acyclic.* □

Theorem 4 *For a join architecture, JCC $\Leftrightarrow$ Comp-C.*

Proof. (only if)
Like in the fork case, levels are trivially found for each scheduler of a join schedule JS: $S_1 \dots S_n$ are of level 2 and S_J is of level 1. The JCC criterion is defined as S_J being CC and the union of ghost-graph and each invoking branch's $\hookrightarrow_{S_i} \cup \rightarrow_{S_i}$ being acyclic (implying that each of the schedules involved is CC). We must show that JS has a level 2, and for this, a level 1 front. A level 1 front is ensured by CC of S_J and the fact that SCC implies Comp-C. What remains to be shown is that a level 2 front exists. To do this, we will merely demonstrate that all steps in the construction are successful (see definition 15).

1. Representing each transaction of level 2 by a calculation on level 1: Because the restriction of the join to one particular S_i branch is a stack, this works for each restricted branch individually (because a stack implies Comp-C). Suppose that it does not work for the global, non-restricted join schedule JS. This would then imply that there is at least one operation of a branch S_i ordered by $<_e$ in between two operations of some other branch S_j, i.e., $\exists o \in O_t, \exists o' \in O_{t'}, \exists o'' \in O_{t'}, t \in T_{S_i}, t' \in T_{S_j}, i \neq j : o' <_e o <_e o''$. However, this would imply the ghost-graph cycle t' $_{JS}$ t $_{JS}$ t', contradicting JCC.

2. Reduction step: once the calculation is found in the previous step, this is guaranteed to succeed.

3. Including the CON and $<_e$ pairs: this poses no problem either.

4. Trivial.

5. Not applicable for joins.

6. This last step is more interesting. It is important to realize that the ghost-graph is included in our notion of $<_e$. In fact, $<_e = \ _{JS} \cup \bigcup_{i=1}^{n} \hookrightarrow_{S_i}$. Hence, JCC implies that $<_e \cup \rightarrow_{S_J}$ is acyclic on constructing a level 2 front. This means CC of this front.

Since each step is possible and successful, there exists a level 2 front. Therefore, JCC implies Comp-C.
(if)
Be a join schedule JS Comp-C, i.e., it has a level 2 and level 1 front. From definition 20 and theorem 1 follows that S_J is level-1-contained in a serial front F_1 and the upper level schedule $S_{up} = (\bigcup_{i=1}^{n} T_{S_i}, \bigcup_{i=1}^{n} \rightarrow_{S_i}, \bigcup_{i=1}^{n} \mapsto_{S_i}, \bigcup_{i=1}^{n} <_{S_i}, \bigcup_{i=1}^{n} \ll_{S_i}, \bigcup_{i=1}^{n} CON_{S_i})$ is level-2-contained in a serial front F_2. So, S_J is level-1-equivalent to a front F_1^* with $\rightarrow_{F_1} \supseteq (\rightarrow_{F_1^*} \cup <_{e_{F_1^*}})$, which implies $\rightarrow_{F_1} \supseteq (\rightarrow_{S_J} \cup <_{e_{S_J}})$ and thus, by definition, S_J is CC.
On the other hand, the upper schedule S_{up} is level-2-equivalent to a front F_2^* with $\rightarrow_{F_2} \supseteq (\rightarrow_{F_2^*} \cup <_{e_{F_2^*}})$, which implies $\rightarrow_{F_2} \supseteq (\rightarrow_{S_{up}} \cup <_{e_{S_{up}}})$. Thus, $\rightarrow_{S_{up}} \cup <_{e_{S_{up}}}$ is acyclic and with $<_e = \ _{JS} \cup \bigcup_{i=1}^{n} \hookrightarrow_{S_i}$ it holds that $_{JS} \cup \bigcup_{i=1}^{n} (\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ is acyclic. Thus, JS is FCC. □