Verifying a Design Pattern for the Fault-Tolerant Execution of Parallel Programs*

Ekkart Kindler Dennis Shasha

Abstract

We present a protocol for the fault-tolerant execution of parallel programs. The protocol leaves the implementation free to make choices concerning efficiency tradeoffs. Thus, we are proposing a design pattern rather than a fully specified algorithm. The protocol is modeled with the help of Petri nets.

Based on the Petri net model, we formally prove the correctness of the design pattern. This verification serves two goals: first, it guarantees the correctness of the design pattern; second, it serves as a test case for the underlying verification technique.

Contents

*Supported by the Deutsche Forschungsgemeinschaft: Project 'Datenkonsistenzkriterien'

1 Introduction

From time to time, computers crash. In most of these cases, the intermediate state of the execution of a program is irrecoverably lost and the program must be restarted from the very beginning. The situation becomes even worse in a parallel program, which is run on different computers. As the running time increases, it becomes more and more likely that the parallel program never terminates successfully.

In order to deal with this unsatisfactory situation, different techniques for recovering from crashes without throwing out all of the program's work have been proposed in the literature. On the one hand, database theory has a long tradition in this field though, originally, recovery concerned only data, not program execution. In databases, stable storage and log files are used for this purpose [4, 16]. On the other hand, there are techniques from distributed computing; e. g. replicated state machines [31] and checkpointing [8]. These techniques, however, cannot recover from a crash of all computers at the same time.

In this paper, we present a protocol for the fault-tolerant execution of long-running parallel programs on a network of computers. The protocol combines techniques from database theory and distributed computing—the protocol tolerates any number of crashes. The protocol is a generalization of the PLinda (Persistent Linda) protocol [20] to conventional parallel programs. In addition, the protocol combines ideas of Lomet and Weikum [26] for transparent application recovery with ideas of Neves et al. [28], which uses *uncoordinated checkpointing*.

The protocol leaves the implementation free to make choices in several areas. For example, it does not determine the *commit points* of the parallel program, on the one hand, but it allows the implementation to insert a commit point any time. Likewise, it does not determine at which time read objects must be released. Other aspects are completely ignored in the model of our protocol. This permits the implementation to tune these parameters to specific efficiency requirements. Therefore, we call the presented protocol a *design pattern for the fault-tolerant execution of parallel programs*.

The focus of this paper is on a formal proof of this design pattern. We prove that the execution of the parallel program on top of the protocol behaves as on a conventional failure-free multiprocessor—even in the presence of crash faults. The notion of a formal proof is vague. We call a proof formal if there is a mathematical model of the protocol, a mathematical representation of the requirement that must be met, and the proof is in terms of this mathematics. We call a proof informal if it is not in terms of this mathematics, but if it is in terms of our understanding of the protocol and of the requirements.

Typically, protocols of this complexity are not verified formally any more because the proofs are long; they are tedious to construct and to read—this applies to our proof too. Nevertheless, we believe that formal proofs are necessary because a proof in terms of our understanding of the protocol may easily miss a point in the proof which is also missing in our understanding of the protocol. This was a problem, for example, in otherwise excellent work by Xu and Liskov [32]. More arguments in favor of formal verification can be found in the book of de Roever et al. [6]. Clearly, informal proofs may help to construct and to communicate a formal proof. But, informal proofs are not sufficient.

In addition to have a verified design pattern for the fault-tolerant execution of parallel programs, the purpose of the proof was to develop and to validate the applicability of a proof method for consistency models. Though still under development, the method used [13, 12, 14, 22, 23] was designed for verifying weak consistency models [7, 11, 17]. Here, we have shown that the proof technique can also be applied to sequential consistency and to a more complex protocol, which combines fault-tolerance, replication, and logging.

There is some other work that deals with the formal verification of DSM-systems. For example, there is a special issue of Distributed Computing [27], which deals with the formal verification of a particular protocol called *lazy caching* [1, 10]. Most methods (e. g. [29, 9, 15]), however, are restricted to sequential consistency or are verified for a limited number of processors only. To the best of our knowledge, there is no formal proof of a protocol which combines fault-tolerance, replication, and logging.

2 The protocol

In this section, we present the protocol for the fault-tolerant execution of a parallel program. Moreover, we characterize the faults that are tolerated by the protocol. As a formal model, we use algebraic Petri nets [30] equipped with different arc-types [23]. For the time being, we can ignore these special features—we will come back to the different arc-types in Sect. 4.1.3.

2.1 The setting

Before introducing the model, we discuss our view of a *parallel program*. We assume that a parallel program consists of a set of *sequential processes* which are executed concurrently. Each parallel program has a distinguished set of *shared variables* that can be accessed by each of its sequential processes. Each shared variable has a unique identifier, and a process can access a shared variable only by this unique identifier.

In the protocol, shared variables are stored on *stable memory*. An access of a process to a shared variable invokes one of the following procedures of the protocol:

```
procedure R(o:identifier):  value
procedure W(o:identifier; v:value)
```

A call `R(o1)` invokes a read operation on the shared variable with identifier `o1`; upon termination, the operation returns some value[1]—informally, this can be considered to be the value of the corresponding variable. A call `W(o1,v1)` invokes a write operation of value `v1` on the shared variable with the identifier `o1`.

Note that there is a difference between a shared variable and its identifier. The identifier is just a name—with no dynamics at all. The corresponding shared variable is a value associated with this identifier. In particular, the associated value can change in the course of time. This difference is obvious, on the other hand; on the other hand, it is subtle and often forgotten. Therefore, we call the set of identifiers of shared variables *objects* in the rest of this paper. And we will use only the term *object* in the formal part of this paper. The dynamic part of a variable is captured by the return values of the read operation.

In addition to the shared variables, each sequential process may have *private variables* that can be accessed by this particular process only. The protocol is not aware of the private variables of a process. The protocol is not even aware when the process accesses a private variable, nor is the protocol aware of the process's internal computations (e. g. a call of an internal procedure along with the involved stack operations or a call of a built-in function) . The protocol is aware only of the process's procedure calls (invocations) of access operations to shared variables.

But, how can we deal with crashes of processes if the protocol is not aware of the private variables, which will be lost upon the crash of a process? The answer is: *continuations*. A *continuation* comprises all parts of a process's state that determine its future behavior. From a given continuation, the computation of a process can be resumed exactly at that point of the computation where the continuation was saved. A continuation of a process typically comprises its code, its program counter, its stack, its heap, and all private variables of the process. Note, however, that the shared variables are not part of the continuation of a process. The shared variables are stored on stable memory, and thus survive crashes. A process can access a shared variable only via the above procedure calls.

Let us introduce a running example of a sequential process in some kind of pseudo code, where the keyword `shared` indicates the declaration of shared variables, and the keyword `private` indicates the declaration of private variables:

[1] For simplicity, we do not consider typing of variables. It will turn out that we need not deal with values at all in the formal part. For providing the correct intuition, we introduce values in this informal presentation.

```
process example;
  shared  x,y;
  private h;

  x:= 1;
  h:= x;
  y:= h
end.
```

Any correct computation of this process will invoke three operations of the protocol: A write of value 1 to object `x`, a read on object `x` returning some value v, and a write of value v to object `y`. Note that there are no invocations of read or write operations for variable `h` because this is a private variable, which is stored and directly accessed by the process itself.

Next, let us consider the values that could be returned by the read operation on `x`. Since the process will be running concurrently with other sequential processes, we cannot tell from the process above which values will be returned by the read operation. From the process's point of view, it could be any value. It will be the responsibility of our protocol to return a value which is *legal* in a computation of this process concurrently running with some other processes. We will give a definition of *legal return values* of a read operation in Sect. 4.2.

Here, we consider only what is *a legal process computation*. Below, there are two computations of the above process, where we represent only those operations seen by the protocol. The return value of a read operation is represented after the read event separated by a colon; remember that the process itself has no influence on this value:

```
W[x,1]  R[x]:1   W[y,1]

W[x,1]  R[x]:27  W[y,27]
```

Both computations are legal for the process `example` above. The first computation is the one that we would expect when the process is run in isolation. But, the second execution could also happen (for example, when another process writes 27 to object `x`).

Next, we show three computations which are not legal (*illegal*) for the process `example`:

```
W[x,1]   R[x]:27   W[y,1]

W[x,27]  R[x]:1  W[y,1]

W[y,1]   R[x]:1  W[x,1]
```

The first one is illegal because the value written to `y` is not the value returned by the read operation on `x`. This does not correspond to the semantics of the private variable `h`. The value returned by the read operation on object `x` and the value written to object `y` should be the same; the process is responsible for properly storing its private variables. The second computation is illegal because the first write event does not write the value 1—the value of the constant in the process's code. Clearly, this is a violation of the process's semantics. The third computation is illegal because it initially invokes a completely wrong event—a write on object `y`.

We do not introduce a programming language and a formal semantics for processes. Consequently, it is impossible to give a formal definition of *legal computation of a process*. We rather assume that, on a path of computation of a process within a given concurrent *execution* of all processes of the parallel program by the protocol, this computation is a legal computation of the process. This will be made precise in Sect. 4.3. For the moment, it is sufficient to know that by computation, we refer to the computation of a sequential process. By *execution*, we will refer to the concurrent execution of all processes of the parallel program by the protocol introduced in the next sections. Within an execution, we will be able to identify the computations of the different sequential processes.

2.2 The parallel program

In this section, we model the *parallel program* on the level of abstraction relevant for the protocol (see discussion above).

Remember that a parallel program consists of a set of *sequential processes* which are running concurrently. From the protocol's point of view, a computation of a single process is a sequence of invocations of read and write operations on some objects (shared variables). The protocol guarantees the persistent and consistent storage of the objects and continuations of the process. In addition, a process can invoke a third operation: a *commit operation*. A commit operation, however, has no effect on the process's semantics. Therefore, we do not require that the invocation of a commit operation is explicitly specified by the sequential process. Commit operations could also be inserted by a compiler or a runtime system because they do not change the process's semantics. Of course, they might affect the performance.

From the protocol's point of view, a process nondeterministically chooses from one of the three operations. This is modeled by the Petri net shown in Fig. 1.

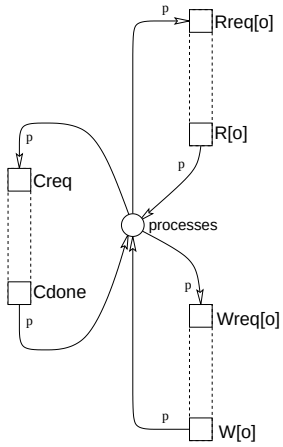


Figure 1: The parallel program

Each operation is split into two parts: a transition that invokes an operation (Rreq[o], Wreq[o], and Creq), and a transition that terminates the corresponding operation (R[o], W[o], and Cdone). These transitions will be merged to the corresponding transitions of the Petri net which models the protocol for the corresponding operation (see Sect. 2.3).

When a processes p has finished one operation and has not yet initiated another operation, there is a token p on place processes. While a token p is on place processes, some internal calculations of p are going on, which cannot be observed by the protocol. Initially, there is no process running since, for simplicity, we assume that all processes are started by the recovery protocol for the server (see Sect. 2.4.6). A process is started by writing its continuation to the database and invoking the restart procedure (see Sect. 2.3.4). For the same reason, all places of the protocol are initially unmarked. They will be initialized during the restart procedure.

2.3 The client's protocols

In this section, we present the protocol for each operation a process may invoke. These protocols are run on the client's side—i. e. on the same processor as the invoking process.

The protocol makes extensive use of locks and is similar to two phase locking [4]: A process must acquire a read lock before reading an object; a process must acquire a write lock before writing an object. Write locks are exclusive, whereas read locks are non-exclusive. A process keeps a write lock until it commits or aborts (i. e. crashes)—as known from strict two phase locking. In contrast to two phase locking, a process may release a read lock at any time—of course, it may not release the lock during a read operation. It

may even acquire a read lock several times before it commits. It will turn out that this is sufficient for guaranteeing *sequential consistency* [25].

The protocol shows two more differences to two phase locking and the classical database approach:

1. When a process accesses an object, it acquires a *local copy* of that object. The local copy is only accessible by this process. In this respect, the protocol corresponds to a *distributed memory model.*

2. The protocol is responsible for restarting crashed processes. Thus, the protocol must also store continuations of processes on stable memory.

In the following sections, we present a Petri net model for each operation that can be invoked by a process. Moreover, we present a Petri net that models the crash of a process, and we present the protocol for restarting a crashed process. Note that these protocols invoke operations of the server. The corresponding protocols on the server's side will be presented in Sect. 2.4; this includes protocols for the *lock manager*, for the *data manager*, for the *log manager*, and for *recovery.* Note that transitions of the client invoking protocols on the server are shaded. The precise meaning of these invocations will be explained in Sect. 2.5.

2.3.1 The read operation

When a process p wants to read some object o, it invokes[2] a read request operation on o as modeled in the process's behavior in Fig. 1. A read request Rreq[o] invokes the protocol shown in Fig. 2: If there is a

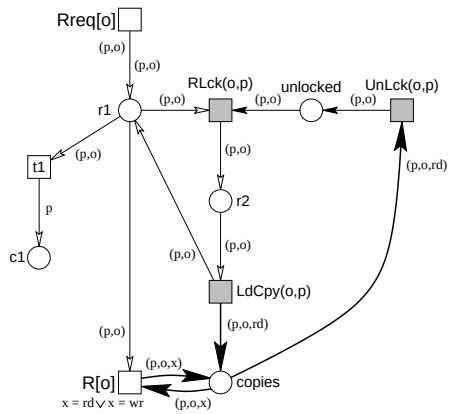


Figure 2: The read operation

local copy of object o, the read event R[o] is immediately executed—terminating the read protocol. If there is no local copy of object o, the protocol first acquires a read lock from the lock manager (RLck(o,p)) and then requests a copy of object o from the data manager (LdCpy(o,p)). Then it proceeds in the same way as described before. Note that place unlocked is marked by a token (p, o) when no local copy of object o for process p is available. Still, place unlocked is initially unmarked because initialization of the protocols on the client's side is part of the restart procedure of the process (see Sect.2.3.4). The protocols for the lock manager and for the data manager will be presented in Sect. 2.4, which discusses the protocols on the server's side.

A local copy of object o for process p is represented by a token (p, o, x) on place copies, where x may, basically, take two values: rd for a copy which was acquired for read only access, and wr for a copy which was acquired for write (and read) access. This way, process p (or rather the protocol running for it) keeps track of its acquired read and write locks. Note that we will have a third value, ud, for x which will be discussed during the commit operation. If a local copy with value wr or rd for x is available, a read operation is allowed on this copy.

[2]This request need not necessarily be explicit established by the read operation; this could also be achieved by appropriate hardware, which invokes the read operation on this object upon a read-miss exception.

As already mentioned, a read lock may be released at any time, when the process has received a copy of the corresponding object. The process releases the read lock on object o by operation UnLck(o,p); at the same time, the local copy of the object is deleted. Note that only copies that were acquired for a read operation can be released by this operation because we explicitly require the third component of the released copy to be rd. In order to allow as much freedom as possible in an implementation of this protocol (or an application of the design pattern), we do not impose any restrictions about when to release a read lock. It depends on the particular application whether it is sensible to keep the lock for some time or whether it is sensible to release the lock right after the read operation.

Note that it might happen that a read operation is stuck while acquiring a lock on some object due to cyclic dependencies in the waits-for graph (see [4]). In that case, we allow the protocol to first execute a commit operation. This is represented by transition t1, which removes a token (p, o) from place r1—indicating the start of a read operation of process p on object o—and adding a token p to place c1—indicating the start of a commit operation (see Sect. 2.3.3).

After the completion of the commit operation, the process may issue the read operation again. Since commit operations do not change the semantics of the program, inserting a commit operation will not do any harm. Since all locks are released during the commit operation, cyclic dependencies can be resolved this way (note that write operations may proceed in the same way). Therefore, the protocol allows resolving cyclic dependencies in the waits-for graph without aborting any process, which is not possible in classical two phase locking.

2.3.2 The write operation

The protocol for a write operation is similar to the protocol for a read operation. The protocol is shown in Fig. 3.

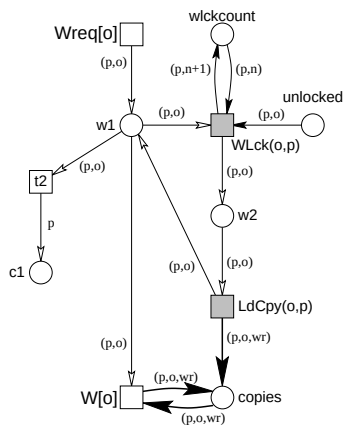


Figure 3: The write operation

For a write operation, it is not sufficient to check the availability of a local copy of the object, since the copy might have been acquired for read access only. A process may access a copy only if it has acquired a write lock for the object. As already mentioned, a copy acquired for write accesses is indicated by attribute wr. If the process has a write lock, it executes the write operation on its local copy immediately.

If the processes has no write lock on object o, it acquires a write lock from the lock manager (WLck(o,p)). At the same time, it increments the number of its write locks (token (p, n) is removed and a token $(p, n+1)$ is added to place wlckcount). Then, it request a local copy of object o from the data manager (LdCpy(o,p)); the local copy receives the attribute wr. Then, the write operation is executed on the local copy.

Note that transition t2 allows the insertion of a commit operation while stuck during acquiring the write lock. See end of Sect. 2.3.1 for more details.

2.3.3 The commit operation

The commit operation writes all changes made by the process since the last commit operation to the *database*. This includes writing the new continuation of the process. During the commit operation, there is a well-defined *commit point*: before the commit point, no changes are made on the database. After the commit point all changes are written to *stable memory*, but are not yet written to the database. If the commit operation terminates successfully (transition Cdone), all changes are also written to the database. However, there could be a crash of the process after the commit point, but before the process could successfully update its changes to the database. In order to deal with this problem, the protocol uses a *log file*, which is also part of the stable memory: The process records all its changes in the log file before the commit point. It is the server's responsibility to update the changes of a process from the log file to the database if a process crashes after its commit point without updating its changes. Altogether, *stable memory* consists of the *log file* and the *database* as known from database theory.

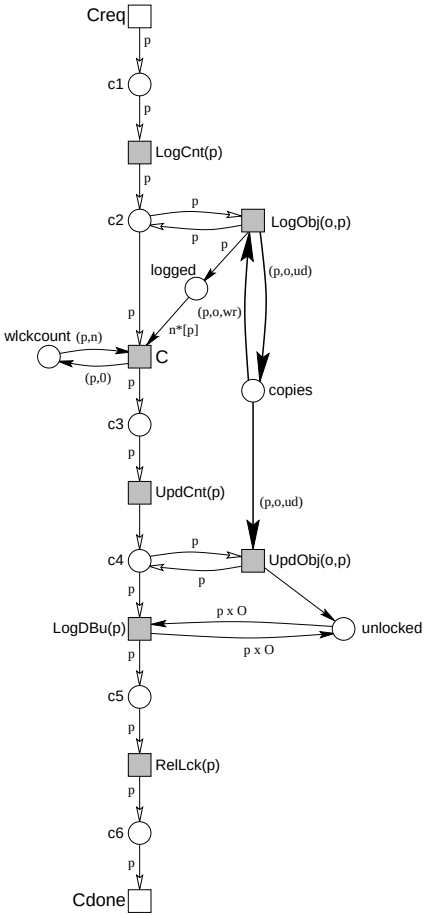


Figure 4: The commit operation

The protocol for the commit operation is shown in Fig. 4: First, the process writes its continuation to the log file (LogCnt(p)) and then writes its local copies of all objects it wrote since its last commit to the log file (LogObj(o,p)). In the model, we do not assume that the copies are written in a particular order to the log file—we just make sure that all copies have been written to the log file before the commit point. This is achieved by keeping track of the number of written objects in the counter n for process p (token (p, n)) on place wlckcount and comparing it to the number of copies written to the log. When all n copies have been written, the process writes a commit record to the log file (C). The successful completion of this write

operation is the commit point during the commit operation.

If the commit record was successfully written to the log file, the continuation and the objects are updated in the database (UpdCnt(p) and UpdObj(o,p)). Upon successful update, the process marks the particular object as unlocked—though still locked at the lock manager. When all copies are marked unlocked—including those that have been read only—we know that we have updated all objects and the continuation in the database. This is recorded by writing a *database updated* record (*DBu* record) for process p to the log file(LogDBu(p)). We will discuss later why the DBu record is necessary. After writing the DBu record, the process releases all its locks at the lock manager (RelLck(p)) and finishes the commit operation.

Note that it is possible that the commit operation crashes right after the commit point. In that case, the database is not updated at all. Since all changes are recorded in the log file before the commit point, the recovery protocols can update the database accordingly (see Sect. 2.3.4, 2.4.4, and 2.4.6).

2.3.4 The fault model and the restart operation

Next, we formalize the fault model for processes. We assume that a process can crash at any time. When a process crashes, it instantaneously loses all its information stored. This includes the processes continuation (i. e. its process variables, its program counter, its stack, etc.) Moreover, it includes data kept by the protocol (i. e. locks, local copies) on the client's side. But, we assume that the crash of a process can be detected.

In the Petri net model, a crash of process p can be represented by a transition that (instantaneously) removes all tokens with a p in its first component (i. e. tokens of the form $(p, ...)$) from all places of the protocols shown in Fig. 1–4 and the protocol for the recovery explained below. In order to detect the crash, a token p is put to the place crashed processes.

The restart of a process is modeled in Fig. 5. When a crash is detected, the process checks in at the server by CheckIn(p). Basically, the check-in defers the restart of the process until the server has redone all committed values of the crashed process and has released all locks held by the process when the crash occurred. The protocol for this redo procedure is discussed in Sect. 2.4.4. After a successful check-in, the process loads its continuation from the data manager by LdCnt(p), initializes the protocol by setting all objects to unlocked and setting the counter for write locks to 0 and restarts the continuation (by placing the continuation to place processes).

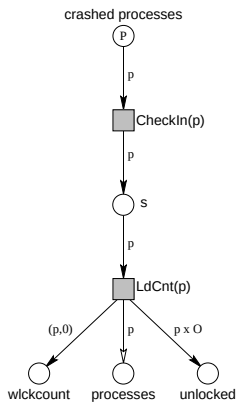


Figure 5: Recovery

2.4 The server's protocols

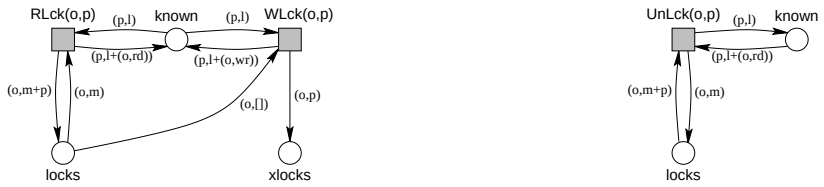
In this section, we present the Petri net models for the *lock manager*, for the *data manager*, and for the *log manager*. The lock manager keeps track of all lock operations of the different processes and guarantees that, if a process holds an exclusive lock on an object, no other process holds a lock (exclusive or non-exclusive) on the same object—or at least not both of these processes successfully commit. Since this property must

be guaranteed even on a crash of the lock manager, it keeps also track of all processes which hold a lock. A process which is not 'known' to the lock manager is not allowed to commit successfully, i. e. to successfully write a commit record to the log file (see discussion in Sect. 3.1).

Since these protocols share information, we assume that they are running on the same processor and, thus, crash simultaneously (see Sect. 2.4.5 for details on the fault model). This restriction can be easily overcome since the only shared information is the set of processes 'known' to the server. This information could be kept by each manager separately. In that case, the protocols must make sure that after a crash of the lock manager, the other managers set all processes to be 'unknown' before the lock manager resumes its normal operation.

2.4.1 The lock manager

Figure 6 shows the protocols for lock and unlock operations. The lock manager keeps two tables: a table for objects, where each object o is associated with a set[3] of processes which have a lock on this object, and a table for processes, where each process is associated with a multiset of objects locked by this process.



(a) Acquiring locks

(b) Releasing individual locks

Figure 6: The lock manager

For each object o, there is a token (o, m) on place locks, where m represents the multiset of processes which have a lock on object o. If this multiset is empty, a process p may acquire an exclusive lock; represented by a token (o, p) on place xlocks. The empty multiset is represented by a pair of square brackets [].

The second table is represented by a token (p, l) on place known, where l is the multiset of pairs (o, x) such that o represents an object and x can take the values rd or wr. A pair (o, rd) in l indicates that process p holds a read lock on o; a pair (o, wr) in l indicates that process p holds a write lock (exclusive lock) on o. The name of the place known indicates another purpose of this list: When there is a token (p, l) on place known the lock manager is aware of the existence of process p (it *knows* process p; see life cycle of a process Fig. 10).

Figure 6(a) shows the protocol for the lock operations and Fig. 6(b) shows the unlock operations for read locks[4]. The unlock operation is the reverse of the corresponding lock operation: we have reverse the arcs.

Figure 7 models the release protocol, which releases all locks held by a process (collective unlock). It moves the token (p, l) from place known to place releasing; in this state, the two transitions below can unlock all locks of process p. When all locks are released, i. e. if the multiset of locked objects is empty, the lock manager marks the process to be known (with an empty list of locked objects).

Note that there are three reasons for a process not to be 'known' by the lock manager: Either the process contacts the lock manager for the first time, the lock manager has crashed, or the lock manager has explicitly decided to ignore the process (see Sect. 2.4.4). There is only one way for a process to become known to the lock manager: invoking a check-in operation (see Sect. 2.4.4). As long as the process is unknown, the lock manager will refuse any other operation. In some cases this implies that the only way for a process to continue is an abort (crash) and a recovery which restarts the process from its latest commit.

[3]In fact, we represent this set as a multiset in the formal model.

[4]The protocol from Sect. 2.3 does only unlock read locks individually; write locks are unlocked collectively during the commit operation by the release operation.

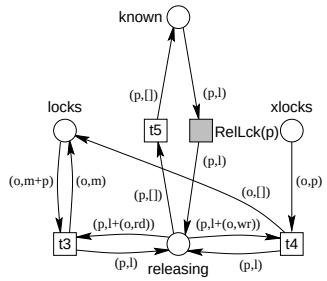


Figure 7: Protocol for the release operation

Note also that some operations of the data manager and the log manager are only available if the corresponding process is known to the lock manager: writing a commit record to the log file and updating an object or a continuation in the database (for details see below).

2.4.2 The data manager

The data manager allows the reading of objects and continuations from the database and the updating of objects in the database. The protocols for these operations are quite simple as shown in Fig. 8. In this model, we have two places which are graphically represented by an ellipse rather than a circle. This indicates that this is stable memory, which is not affected by crashes. Tokens on these places will not be removed by a crash—this will be formalized in the fault model for the server. From the Petri net point of view, the ellipses do not make any difference—the graphical distinction should help to spot stable memory in the protocol more easily.

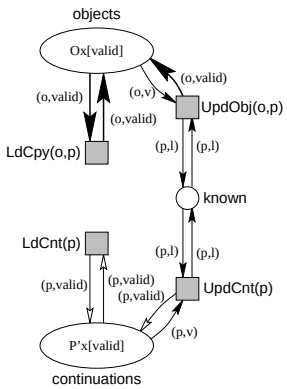


Figure 8: The data manager

The database objects initially stores a value for each object (shared variable of the program); the database continuations initially stores the continuations of all processes. Both load operations, LdCpy(o,p) and LdCnt(p), return the requested copy respective continuation but leave the database unchanged. We assume that the update operations UpdObj(o,p) and UpdCnt(p) overwrite the old version in place. Both update operations can only be initiated by a process when the process is known to the server. The load operations, however, may be executed any time.

Since writing to disk is not reliable, it may happen that we overwrite an old value without having the new value in that place. This will be formalized in the fault model. We can recover from such faults by the help of the information in the log file. This, however, requires that we are aware of a write fault. In order to detect write faults, the protocol not only writes the object itself but also a checksum. Moreover, we assume that a write fault always results in an invalid checksum (see fault model in Sect. 2.4.5). In our model, we do

not explicitly represent the checksum. We add a second two-valued component to the object written, where *valid* stands for a correct checksum and *invalid* stands for an incorrect checksum of this particular object or continuation. A load operation will return a value only if the corresponding checksum is *valid*. Otherwise, the operation will be blocked—it can be resumed only by invoking the restart procedure (see Sect. 2.4.6). Of course, an update operation may overwrite any object–even if *invalid*. This is reflected by a variable v for the validity flag in the corresponding arc-inscription. Since variable v can take any value, updating is possible in any case.

2.4.3 The log manager

The log manager allows writing to the log file, which is part of stable memory. There are four kinds of log records: *update record*, *continuation records*, *commit record*, and *database updated record*. An update record consists of a pair (o, p). Its meaning is that process p has written object o. A continuation record (cnt, p) is a continuation of process p. A commit record (cmt, p) indicates the successful commit of process p. A database updated record (dbu, p) indicates that a process has updated the database after its successful commit. In order not to confuse update records and database updated records, we will call database updated records *DBu* records in the rest of this paper.

In addition, log records receive consecutive sequence numbers. Moreover, there is a counter which points to the sequence number of the last log record written. This counter is called *log counter* and is also held on stable storage.

The protocols for these three operations are quite simple and are shown in Fig. 9. The log counter is incremented and the corresponding log record is written with this increased sequence number. To write a commit record or a DBu record, the process must be known to the lock manager. Otherwise, the process which invokes this operation is blocked—it can resume operation only by aborting and recovering from the latest commit. The reason for this blocking will be discussed in Sect. 3.1.

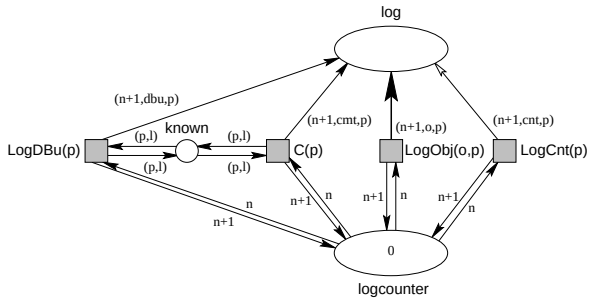


Figure 9: The log manager

2.4.4 The check-in and the redo procedure

In Sect. 2.3.4, we have seen that a process and the protocols running on the client's side may crash. Upon crash, the state of the process and the corresponding protocols at the client's side is lost. Only the information that was explicitly saved on stable memory by invoking the corresponding sever operations will survive a crash.

After a crash of a process, the server must make sure that the values in the database are consistent. In particular, it must make sure that the values of the log file are updated to the database if the process crashed right after successfully writing the commit record to the log file. On the other hand, the server must make sure that no values are updated if the process crashed before the commit point. This is the task of the *redo procedure* for a single process. Moreover, the server must make sure that the process cannot be restarted before the redo procedure is completed. Otherwise, it could happen that it starts from an old continuation. Likewise, the server must guarantee that other processes access the variables written by the crashed process only after the redo procedure.

The Petri net from Fig. 10 shows the life-cycle of a process from the server's point of view: Initially *unknown* (after a successful start of the server), a CheckIn(p) makes the process *known* to the server. Remember, that some crucial server operations can only be invoked by a process (e. g. a writing a commit record) that is known by the server. The server may decide to ignore a known process at any time. Typically, the server will decide to ignore a process if the server has some evidence that the process has crashed[5]. In the protocol, however, the server can ignore a process at any time. This way, the protocol and, in particular, its correctness proof are independent from a particular crash detection mechanism. Moreover, the server is free to ignore any process if this appears to be appropriate for the over-all execution (e. g. for performance reasons).

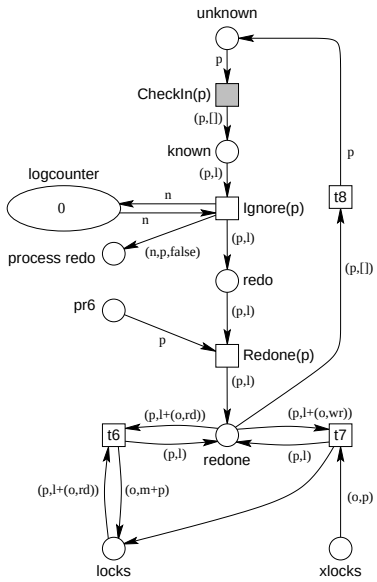


Figure 10: Life cycle of a process from the servers point of view

When the server decides to ignore a process, it first starts a redo procedure for this process in order to make sure that the database is in a consistent state. The redo procedure is modeled in the Petri nets from Fig. 11–14 and will be explained below. After a successful termination of the redo procedure, the server releases all locks held by the process in the same way, locks are released by the process's release operation (cf. Fig. 7). When all locks are released, the process is *unknown* at the server. This completes the life-cycle of a process at the server.

Next, we present the redo procedure for a process. After the server decided to ignore a process (occurrence of transition Ignore(p); see Fig. 10), the redo procedure scans the log file backwards. If it encounters a commit record for the process first, it writes the subsequent update records to the database. When it encounters a continuation record of the process it writes the continuation to the database, and then terminates the redo process by writing a DBu record to the log file in order to make sure that a process is not redone twice (see Sect. 3.2 for a detailed discussion). If the first encountered log record for the process is no commit record, the redo procedure terminates immediately—without updating the database and writing a DBu record. Note that, during the redo procedure of a process, the log records of all other processes are ignored.

Let us consider the protocol for the redo procedure in some more detail. Figure 10 shows the start of the redo procedure: transition Ignore(p) puts a token $(n, p, false)$ to place process redo. The token $(n, p, false)$ on place process redo says that currently a redo procedure for process p is in progress, that the next log record to be read is the one with sequence number n (*scan counter*), and that a commit record for p has not yet been encountered ($false$). A *true* in the third component (*commit flag*) will indicate that a commit

[5]Remember, that process can resolve cyclic waiting for locks themselves by establishing an early commit; therefore, there is no need for the server to resolve cyclic wait situations.

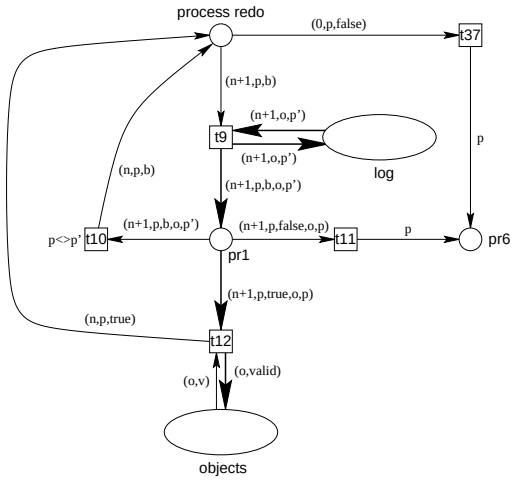


Figure 11: Redo p: Operations for an update record

was encountered during the backwards scan. Initially, the scan counter is set to the value of the log counter, which represents the highest sequence number of a log record in the log file. A token p on place pr6 indicates the termination of the redo procedure. Then, the server can proceed to release the locks of process p.

Figure 11 shows the actions taken when an update record is encountered during the backwards scan: If it is an update record of a different process p', the record is ignored and the scan counter is decreased. If it is an update record of process p and a commit was encountered (i. e. the commit flag is $true$), the corresponding object is updated to the database (occurrence of transition t12)). If no commit record has been encountered (i. e. commit flag is $false$), the redo procedure for p terminates immediately (occurrence of transitions t11). Figure 11 shows also the action taken when the backwards scan reaches the end (or rather the begin) of the log file: The occurrence of transition t37 terminates the redo procedure. Note that this can happen only if no commit was encountered during the backwards scan.

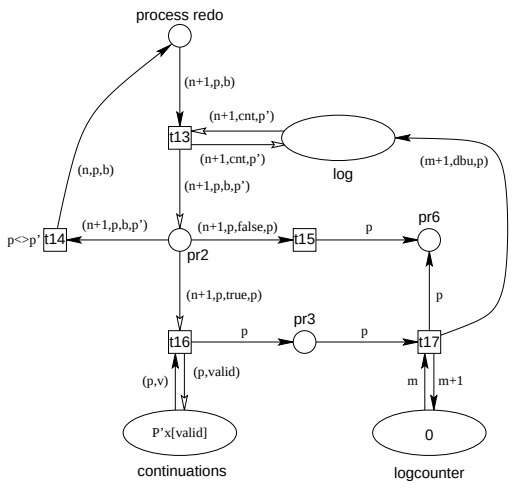


Figure 12: Redo p: Operations for a continuation record

Fig. 12 shows the actions taken, when a continuation record is encountered during the redo procedure. Again, the log record is ignored if it is the log record of a process p' different from p. If it is a continuation record for process p and a commit was encountered, the continuation is updated to the database (transition t16), a DBu record is written to the log file (transition t17), and the redo procedure terminates. If the

continuation record is encountered before a commit, the redo procedure terminates (transition t15)—without updating the database and writing a DBu record.

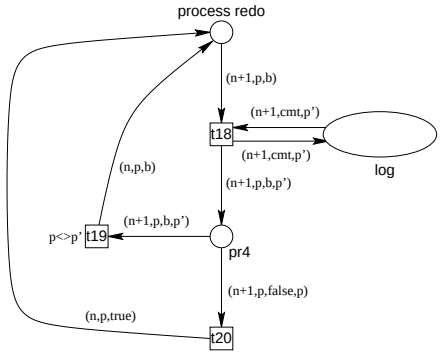


Figure 13: Redo p: Operations for a commit record

Figure 13 shows the actions taken, when a commit record is encountered. Again, a commit record of a process p' different from p is ignored (transition t19). If it is a commit record for process p, the commit flag is set to *true* (transition t20). Note that it cannot happen that a commit record is encountered when the commit flag is true already.

At last, Fig. 14 shows the actions taken, when a DBu record is encountered. Again, a DBu record of a process p' different from p is ignored (transition t22). If a DBu record for p is encountered, the redo procedure terminates (transition t23).

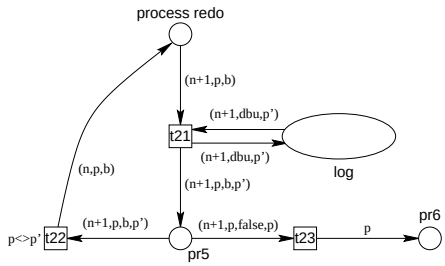


Figure 14: Redo: Operations for a DBu record

2.4.5 The fault model

As mentioned previously, the server (lock manager, data manager, and log manager) must be able to deal with two kinds of faults: A crash of the processor on which the server is running (*server crash* for short) and a write fault on the database. A crash of the server results in a loss of all its volatile data but does not affect the stable memory (database, log file, and log counter). A write fault results in an object with an invalid checksum in the database.

The crash can be modeled in the same way, as a crash of a process. A transition, which instantaneously removes all tokens from all circle-shaped places of the models shown from Fig. 6 to Fig. 14 and even the circle shaped-places of the recovery protocol defined below. Again we assume that, upon crash, a specific place server crash will be marked—which allows us to detect this crash (see below).

The write faults of the data manager are modeled in Fig. 15. It provides another instance for transitions UpdObj(o,p) and UpdCnt(p) that do not correctly write the object or the continuation[6]. By assumption, we know that in this case, the checksum is invalid.

[6]Not correctly writing the object or the continuation is represented by using a non appropriate arc type, which still has to be introduced (see Sect. 4.1.3)

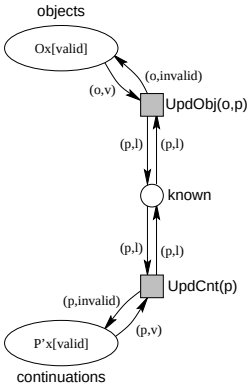


Figure 15: Write faults

2.4.6 Server recovery

When the server is restarted after a crash, it first scans the log file from the highest sequence number to 0 in order to update all objects which have been written (and committed), but have possibly not been updated in the database due to the crash. Note that we only need to redo operations but do not need to undo operations because updates to the database will be only made after the commit point.

When the backwards scan is finished, the lock table is initialized (no object is locked) and the table of known processes is initialized (all processes are unknown). Then, the server is ready to accept requests for the lock manager, the data manager, and the log manager.

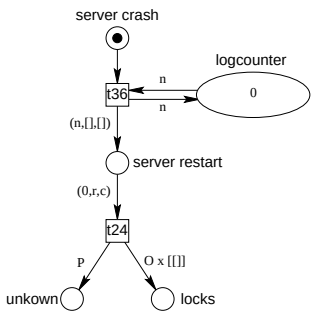


Figure 16: Recovery of the server: Start backwards scan and initialize tables

Figure 16 shows the initiation of the backwards scan, by reading the log counter and initializing the table of objects and continuation which have been updated during the backwards scan and a table which tells which processes have committed during the scanned section of the log file (see below). The scanning of the log file itself and the operations taken when a particular log record is encountered are modeled in Fig. 17–Fig. 19 below. The backwards scan terminates, when the *scan counter* is 0. Then, the tables of the server are initialized.

A token (n, r, c) on place server restart says that n is the sequence number of the next scanned log record (*scan counter*). The second component, r, is the set of all objects and continuations which have already been updated during the current scan of the log file (*redone list*). The third component, c, is the set of all processes for which a commit record was encountered during the current scan (*commit list*). Initially, the redone list r and the commit list c are empty.

Figure 17 shows the actions taken when an update record for object o of process p is encountered (i. e. a record $(n + 1, o, p)$). If the process is not in the commit list (i. e. $c[p] = 0$), this record is ignored and the scan counter is decreased by one. If the object is in the redone list, the log record is also ignored and the scan counter is decreased. If, however, the process is in the commit list and the object is not in the redone

list (i. e. $r[0] = 0$), the object is written to the database, the scan counter is decreased, and the object is added to the redone list.

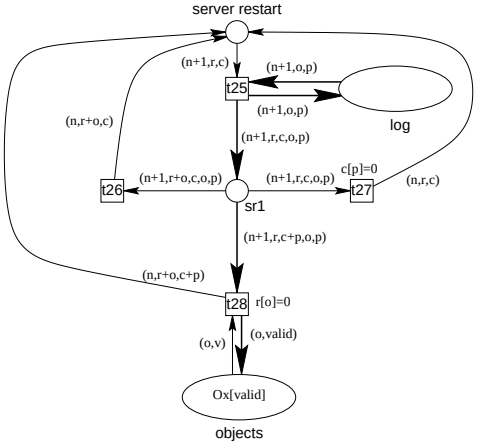


Figure 17: Operations for an update record

Figure 17 shows the actions taken when a continuation record for process p is encountered (i. e. a record $(n+1, cnt, p)$). If the process p is not in the commit list, the record is ignored and the scan counter is

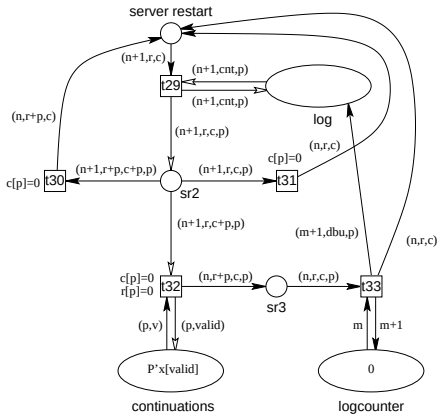


Figure 18: Operations for a continuation record

decreased. If the process is in the commit list, but the process is also in the redone list, the record is also ignored, and the scan counter is decreased. In addition, however, process p is removed from the commit list because a continuation is the first record written to the log file during the commit operation. If the process is in the commit list and is not in the redone list, the continuation of p is updated to the database. Then, the process p is removed from the commit list and added to redone list. Moreover, the scan counter is decreased and a DBu record for process p is written to the log file.

Figure 19 shows the action taken when a commit record for process p is encountered (i. e. a record $(n+1, cmt, p)$). The process is added to the commit list and the scan counter is decreased. Note that a commit record for process p cannot be encountered when the process is already in the commit list; therefore, we require $c[p] = 0$ in that situation. If this is not true, some unforeseen faults (a fault not covered by or fault model) has occurred. In that case, the server cannot successfully recover.

At last, Fig. 20 shows the action taken when a DBu record for process p has been encountered (i. e. a record $(n+1, dbu, p)$). During the restart of the server, these records are ignored—DBu records are relevant only for redoing a single process (see Sect. 2.4.4).

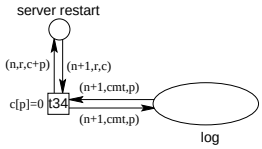


Figure 19: Operations for a commit record

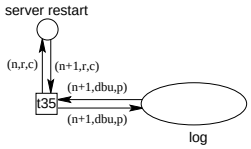


Figure 20: Operations for a DBu record

2.5 The combined model

In Sections 2.2 through 2.4, we have presented many Petri nets, which modeled different aspects of the protocol. Though intuitively clear, it remains to show how the different Petri nets interact with each other. To this end, we combine all Petri nets into single large Petri net, which we call the *combined Petri net model*. In the formal proof, we will refer to this combined Petri net; fortunately, it is not necessary to draw it as a single Petri net—we just show how it is constructed from the different Petri nets.

Technically, the combined Petri net model is the union of all places and transitions of the different Petri nets, where the shaded transitions of the client are synchronized with a corresponding shaded transition (one with the same name) of the server. Since this is a standard construction in Petri net theory, we confine ourselves to an informal explanation of the synchronization operation. In our model, we have three cases:

1. There is exactly one transition at the clients side and one transition at the servers side which carry the same name. In this case, there is exactly one transition with this name in the combined Petri net model. Figure 21 shows how this transition looks like for the synchronization of the RLck(o,p) transitions. On the top of this figure, we have shown the client's transition and the server's transition. On the bottom, we have shown the transition of the combined Petri net model. This transition inherits all arcs from the two original transitions.

   The same construction applies to the following transitions: WLck(o,p), UnLck(o,p), C, LdCnt(p), LogObj(o,p), LogCnt(p), LogDBu(p), and CheckIn(p).

2. There are two different transitions on the client's side that both invoke the same operation. In our model, LdCpy(o,p) is the only example. This transitions is present in the Petri net model for the read operation as well as in the Petri net model for the write operation. In that case, we have two instances of this transition in the combined Petri net model.

   The three original transitions (two on the client's side and one on the server's side) are shown at the top of Fig. 22. The two different instances of the corresponding transition in the combined model are shown at the bottom of Fig. 22.

3. At last, we have the case that there is only one invoking transition at the client's side, but there are two corresponding transitions on the server's side. Transitions UpdObj(o,p) and UpdCnt(p) are examples for this case. Note that the second instance on the server's side originates from the fault model. An UpdObj(o,p) can either be successful or non successful. Symmetrical to the above case, we have two instances of this transitions in the combined Petri net model. Figure 23 shows the three UpdObj(o,p) transitions of the original Petri net model on the top and the two instances of this transition in the combined Petri net model on the bottom.

In order not to clutter the Petri net models with shaded transitions, we did not shade the transitions of the program model that invoke and terminate the read, write, and commit (cf. Fig. 1 and the models

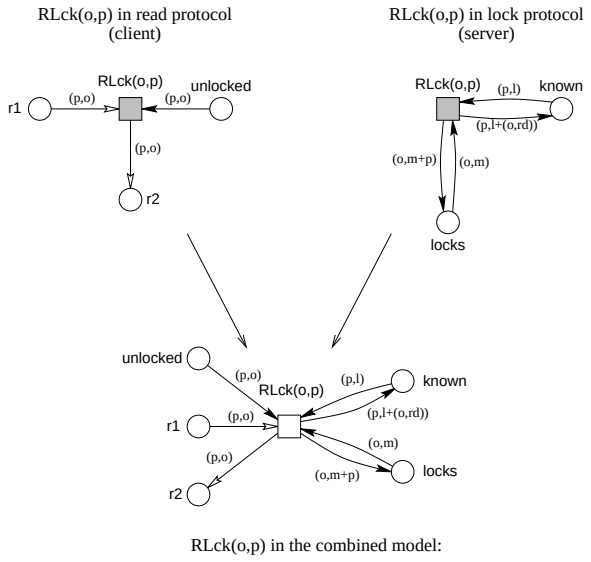


Figure 21: Synchronization of the RLck(o,p) transitions.

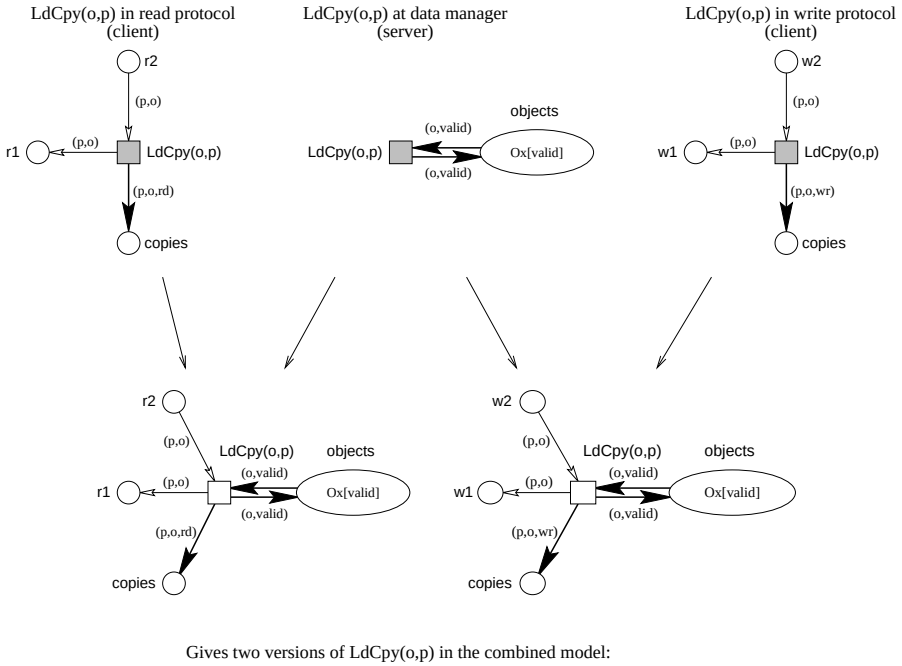


Figure 22: Synchronization of the LdCpy(o,p) transitions.

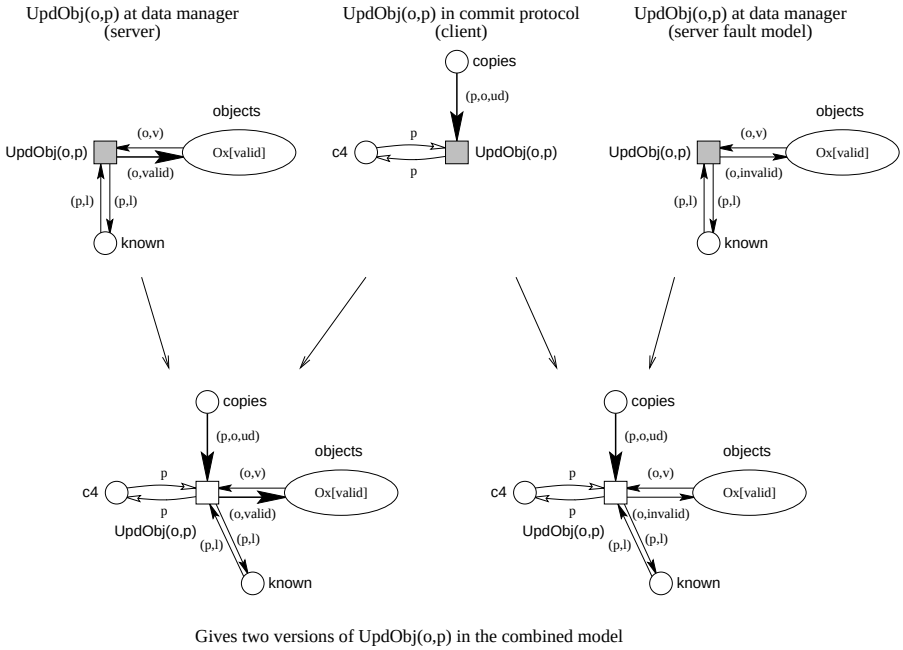


Figure 23: Synchronization of the UpdObj(o,p) transitions.

from Sect. 2.3). For each transition Rreq[o], R[o], Wreq[o], W[o], Creq, and Cdone, there exists exactly one transition in the Petri net from Fig. 1 and exactly one in one of the Petri nets from Figures 2–4. Thus in the combined Petri net model, there will be exactly one transition for each of these pairs. The construction is the same as in the first case (cf. Fig. 21).

2.6 The data types

For completeness sake, Table 1 defines the domains and operations that are used in the Petri net mode. Moreover, this table associates each variable that occurs in the Petri net with a domain. Altogether, we call this the data type of the Petri net model.

Table 2 list the *token domain* associated with each place. A legal marking associates each place with a multiset over its token domain. In a Petri net, an inscription of an arc to or from a place must evaluate to a multiset over the place's token domain. For convenience, we allow one exception: If the inscription of the arc evaluates to the an element of the place's token domain (rather than a multiset over the token domain), we implicitly consider it as a singleton multiset (the corresponding operation is called multiset injection; cf. Fig. 2). Since most arc inscriptions represent a singleton multiset, this convention helps to avoid cluttering arc inscriptions with multiset injection operations $[a]$ instead of a.

3 Discussion of the protocol

In this section, we discuss some subtle points of the protocol in order to motivate some design decisions of the protocol and in order to demonstrate what could go wrong.

3.1 The need for restricting invocation of some operations

At first, we have a closer look to the distinction of *known* and *unknown* processes at the server. This distinction was motivated by the following assumption: We assume that the server has no means to distinguish a slow process from a crashed process. Thus, the server cannot tell whether a server has crashed after a

Domains:	$process$	any set
	$object$	any set disjoint from $process$
	$bool$	$= \{true, false\}$
	nat	the natural numbers including 0
	$mode$	$= \{rd, wr, ud\}$
	$checksum$	$= \{valid, invalid\}$
	$logtype$	$= \{cnt, cmt, upd\} \cup object$
	$MS(process)$	multiset over $process$
	$MS(object)$	multiset over $object$
	$MS(object \times mode)$	multiset over $object \times mode$
	$MS(process \cup object)$	multiset over $process \cup object$
Variables:	p, p' : $process$	
	o : $object$	
	n : nat	
	b : $bool$	
	x : $mode$	
	v : $checksum$	
	m : $MS(process)$	
	l : $MS(object \times mode)$	
	c : $MS(process)$	
	r : $MS(process \cup object)$	
Constants:	P : $MS(process)$, multiset in which each process occurs exactly once.	
	O : $MS(object)$, multiset in which each object occurs exactly once.	
	$0, 1$: nat, with their usual meaning.	
Operations:	$(\cdot, \ldots, \cdot)$	tupeling
	$\cdot + \cdot : nat, nat \to nat$	addition on natural numbers
	$\cdot + \cdot : MS(D), MS(D) \to MS(D)$	multiset addition on $MS(D)$ for any domain D
	$[\cdot] : D \to MS(D)$	multiset injection
	$\cdot * \cdot : nat \times MS(D) \to MS(D)$	scalar multiplication on multisets

Table 1: Data type: Domains, variables, constants, and operations

processes	: $processes$
r1, r2, w1, w2	: $process \times object$
c1, c2, c3, c4, c5, c6	: $process$
unlocked	: $process \times object$
copies	: $process \times object \times mode$
wlckcount	: $process \times nat$
crashed processes, s	: $processes$
known, releasing	: $process \times MS(objects \times mode)$
redo, redone	: $process \times MS(objects \times mode)$
locks	: $object \times MS(process)$
xlocks	: $object \times process$
continuations	: $process \times checksum$
objects	: $object \times checksum$
log	: $nat \times logtype \times process$
logcounter	: nat
unknown	: $process$
process redo	: $nat \times process \times bool$
pr1	: $nat \times process \times bool \times object \times process$
pr2, pr4, pr5	: $nat \times process \times bool \times process$
pr6	: $process$
server crash	: dot
server restart	: $nat \times MS(object \cup process) \times MS(process)$
sr1	: $nat \times MS(object \cup process) \times MS(process) \times object \times process$
sr2	: $nat \times MS(object \cup process) \times MS(process) \times process$

Table 2: Token domains for places

commit if there are no subsequent update operations of this process within a given time limit—the process might just be slow for some reason. In order not to block other processes forever, the server might withdraw the locks from a process that appears to have crashed—even if the server can never be sure that the process has crashed.

Figure 24 shows such a scenario. Process $p1$ writes to some object $o1$. During the commit operation, the process writes an update record for $o1$ to the log file and then successfully commits. Let us assume that $p1$ crashes right after the commit point (i. e. after the successful write of the commit record). This part of the execution is represented on the left hand side of Fig. 24—event t12 updates the update record of process $p1$ to the database. Since the state of a process and the state of the protocol running on the client's side are lost upon a crash, only the server has enough information to bring the database to a consistent state. Thus, the server must make sure that the value of $o1$ written by $p1$ is updated to the database. This is achieved by ignoring process $p1$ at some point and starting the redo procedure for $p1$. This part of the execution is represented in the middle part of Fig. 24. After the server has released the locks of $p1$ on object $o1$, another process may access object $o1$. Let us assume that process $p2$ first reads $o1$, then writes to $o1$, then successfully commits, and then updates the value of $o1$ to the database. This part of the execution is represented on the right hand side of Fig. 24.

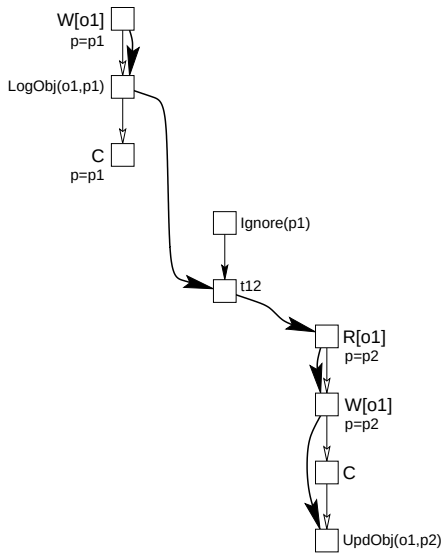


Figure 24: Database updated by server after a crash of $p1$

So far, nothing bad has happened. The scenario from Fig. 24 just shows that the server must eventually update the value written by process $p1$ to the database if $p1$ crashes right after the commit point. Now, let us consider a slightly different scenario: Process $p1$ did not crash after the commit operation but just was slow. Remember that the server has no means to distinguish this scenario from the scenario before. So, it will eventually decide to ignore process $p1$ and to update the values written by $p1$ to the database. Now, let us assume that processes $p1$ eventually invokes the update operation for object $o1$—after process $p2$ has updated its value to the database. This scenario is shown in Fig. 25: It is a lost-update situation because the value written by process $p2$ after reading the value written by $p1$ is overwritten by the value of $p1$ again. The value of $o1$ written by $p2$ is lost. Clearly, we do not want to allow such lost-update situations. Therefore, the server must make sure that a process cannot update its written values to the database once the server has decided to ignore it. For that reason, the protocol allows a process to invoke an update operations only, when it is *known* to the server. After deciding to ignore a process, the server does not know the process any more—after some intermediate states which are needed to redo the process and release all its locks.

Now, the question is which are those crucial operations that can be invoked only by a known process? For example, the protocol allows to load objects from the database even by unknown processes. Likewise, an

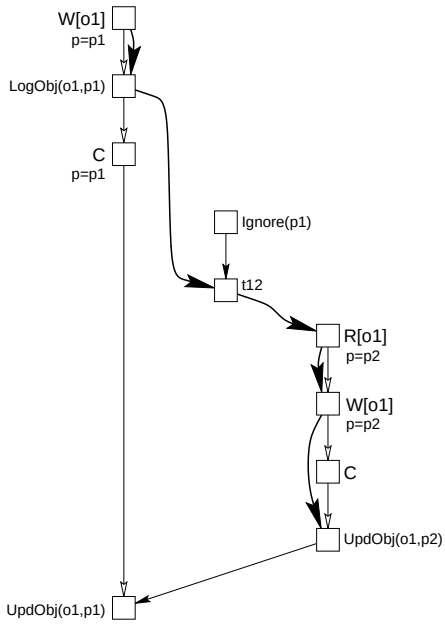


Figure 25: Database updated by server and by a slow update operation of $p1$

unknown process may write update and continuation records to the log file. The answer to this question can be found in the protocols of the server: All operations with an arc from place known to the corresponding transition can be invoked only by a known process.

But, there remains a more difficult question: How can we ever be sure that we did not forget another scenario which results in a lost update or in another undesirable effect? We will come back to this question in Sect. 3.3.

3.2 The need for database updated records

Next, let us discuss the following questions: Do we need DBu records at all? In which situations must a DBu record be written?

Let us start with answering the first question and assume that the protocol does not write DBu records at all. From the scenario of Fig. 24 we already know that the server must update the value written by $p1$ to the database after a crash of $p1$. Now, let us assume that process $p1$ is fast and updates $o1$ to the database itself, then releases its locks, and crashes right after the release operation. Moreover, let us assume that process $p2$ reads object $o1$, then writes to $o1$, then commits, and eventually updates $o1$ to the database as shown in Fig. 26. This time, the server is slow and starts to ignore $p1$ after $p2$ has updated its values to the database. Since $p1$ has not written any further log record before it crashed, the server will find the very same log records as in the scenario from Fig. 24 (remember that we currently consider the protocol without DBu records). Thus the server will take the same actions during the redo procedure for process $p1$: It will update the value of $o1$ written by $p1$ to the database. Again, we have a lost-update situation. The reason is that the server did not know from the log file that process $p1$ has already updated its values to the database and released its locks. Therefore, the protocol requires that the process writes a DBu record after updating all its values to the database and before releasing its locks—the DBu record allows the server to detect that $p1$ has already been updated the value of $o1$ itself.

This answers the question whether we need DBu records. This leaves the question whether it is sufficient to write a DBu record after the update operation of the process. In fact, it turns out that this is not sufficient. Scenarios similar to the above one show that, whenever some updates to the database have been made (by the process itself, by the server during a redo of the process, or by the server during its restart

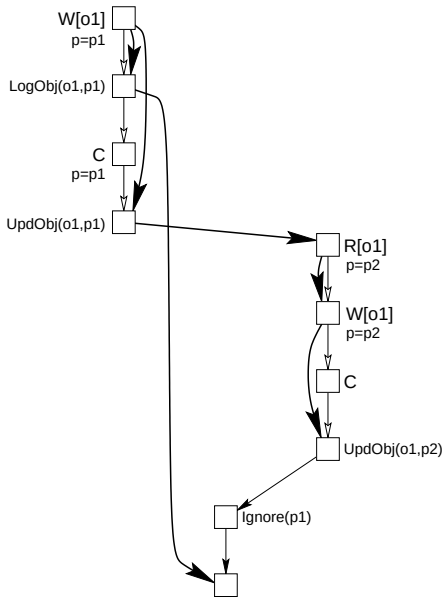


Figure 26: Another lost update situation: Late update by the server

procedure), a DBu record for the corresponding process must be written to the database.

Again, we are left with the question: How can we be sure that we did not forget another scenario which results in a lost update or in another undesirable effect?

3.3 The need for formal methods

Let us come back to the question posed in the previous section: Can we be sure that we did not forget a scenario which is undesirable? In the database and the transaction community, there is a lot of understanding on the undesirable situations and on what could go wrong. In our protocols, however, the situation is different from the classical database situation:

1. There are continuations of processes, which must also be stored in the database; in this sense, processes take a similar role as objects in the classical situation. On the other hand, processes take a similar role as transactions in the classical situation. This dual role of processes does not occur in the classical database situation.

2. Though processes are similar to transactions, there are some crucial differences. First of all we do not require that a process terminates (think of an infinite computation that calculates the digits of π). Moreover, a single process may have many commit events. It might even happen that we have a commit event and later on an abort (crash) of the same process—this does not occur in the classical database situation[7].

On the one hand, the understanding of classical transaction theory might help to identify critical scenarios. On the other hand, it might provide a wrong intuition concerning processes. Therefore, we claim that the only way to be sure not to forget some cases is to provide

1. a specification of what correctness should mean. In particular, the specification must be a precisely defined mathematical object, which characterizes those operational models that satisfy the specification and those that do not satisfy the specification. In that case, we call it a *formal specification*.

[7] One might argue, that processes are similar to nested transaction; but, nested transactions do not faithfully capture the concept of processes.

2. Moreover, we must provide a proof that the operational model satisfies the formal specification. In order not to forget cases, the proof must be in terms of the operational model of the protocol rather than in terms of an informal understanding. In that case, we call it a *formal proof*.

In Sect. 4 we will present a formal specification of correctness and in Sections 5, 6, and 7, we will present a formal proof that our Petri net model (the combined Petri net model defined in Sect. 2.5) of the protocol satisfies this specification.

4 Specification

Informally, the protocol should guarantee that a parallel program which is executed on a faulty system behaves as on a non-faulty system with a conventional shared memory. But what precisely does this mean? For example, the protocol cannot guarantee any real-time properties for the program on the faulty system since we did not impose any assumption on the frequency and the duration of crashes. Moreover, we need to express the above informal requirement in terms of the Petri net model—or rather in terms of the executions of the Petri net model—in order to give a formal proof.

Therefore, we first have a closer look to the executions of the Petri net model before formalizing correctness. Correctness will be defined in two steps: First, we define the *correctness of an execution*. Second, a Petri net model is *correct* if all its executions are.

4.1 Executions

In this section, we discuss the executions of the Petri net model.

4.1.1 Petri net executions

Figure 27 shows one *execution* of the Petri net model, where we assume that there is only one process $p1$ and one object $o1$ around. Note that, in an execution, we abbreviate the names of the places from the Petri net. For example, svcr stands for server crash, cr stands for crashed processes, cn stands for continuations, and ob stands for objects. A list of all abbreviations can be found in Table 3 on page 104.

Remember that, initially, the server as well as all processes are not running and thus need to be restarted from the database. First, the server is restarted. Since the log counter (abbreviated lc) is 0 initially, the back-scan procedure terminates immediately and the table of locks (for the only object $o1$) is immediately initialized (lck.(o1,[])). Moreover, process $p1$ is unknown at the server right after this initialization (uk.p1). Then, process $p1$ checks in and loads its continuation (cn.p1) and restart it (p.p1). Upon restart, the write lock counter of $p1$ is initialized to 0 (wlc.(p1,0)) and all objects (here only object $o1$) are marked as unlocked (unl.(p1,o1)). Then, the process issues a write request on object $o1$ (Wreq[o1]). Since no local copy of this object is available, a write lock operation is issued. The write lock operation increments the write lock counter to 1 on the process's side and marks object $o1$ as exclusively locked by process $p1$ on the server's side (xlck(o1,p1)). Then, the process loads a local copy of object $o1$ from the server and sets the write attribute on this copy (cp.(p1,o1,wr)). After this, the write operation is performed on the local copy—finishing the execution of the write operation. Then, the process issues a read request on this object. Now that there is a local copy of that object, the read can be immediately executed on this copy. Then, process $p1$ could execute the next operation and so on. We do not consider the execution beyond this point.

Of course, there are many other executions—in fact, there are infinitely many others. For example, process $p1$ could start with a read operation. And there are even more possibilities when more than one process and more than one object are involved.

Technically, an execution is a so-called *non-sequential process* of a Petri net [5] which is equipped with some additional information. Basically, an execution is an unwinding of the Petri net model starting in its *initial marking* of the Petri net. It is again a Petri net—a Petri net without cycles and with non-branching places. In order to make a clear distinction between a Petri net model and its execution, we call a transition in an execution an *event* and we call a place in an execution a *condition*. A condition is said to be an *initial condition* if it has no preceding event. In our example, these are the top-most conditions. The set of initial conditions corresponds to the initial marking of the Petri net. Note that, in an execution, different events may correspond to the same transition of the Petri net, and different conditions may correspond to the same token on the same place. We say that the different events represent different occurrences of the same transition and the different conditions represent different occurrences of the same token on the same place.

In addition to the classical definition of non-sequential processes, we incorporate more information in the Petri net and its execution. We have distinguished among different arc-types in the Petri net, which immediately carry over to the executions. Moreover, we have inscriptions of transitions. The meaning of

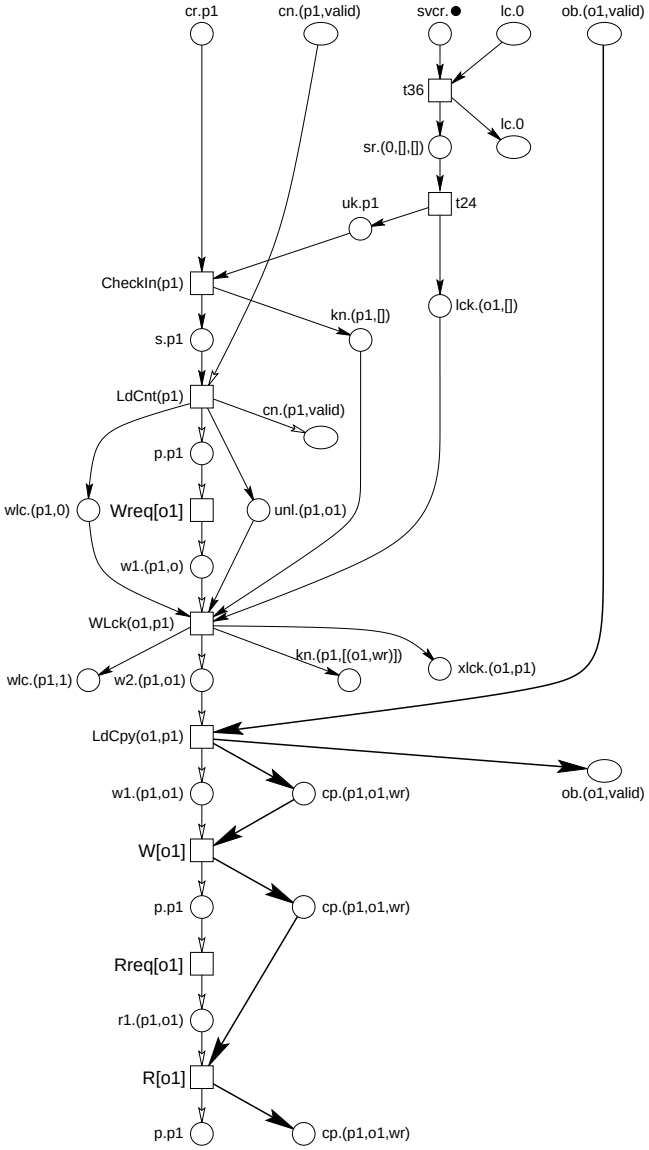


Figure 27: An execution

the different arc-types will be discussed in Sect. 4.1.3. A formal definition of these so-called *Arc-typed Petri Nets* and their executions can be found in [23].

4.1.2 Transactional view

The example execution from Fig. 27 already indicates that, even for simple scenarios, the corresponding execution of the Petri net model is quite large. The reason is that this execution captures all details of the protocol. Fortunately, we need not deal with all these details in the specification. Therefore, we introduce *abstract views* of an executions masking those details that are not relevant for a particular purpose. For example, Fig. 28 shows the essence of the execution from Fig. 27 from a transactional point of view: a

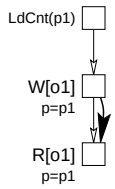


Figure 28: Transactional view

continuation of process $p1$ is loaded from the server and started. Process $p1$ issues a write and a read operation on object $o1$. The read operation returns the value written by the write operation—indicated by the bold-faced arc from the write event to the read event. Since we omitted the conditions in the immediate context of each event, we cannot tell from the event itself to which process it belongs. Therefore, we explicitly state the involved process by a label $p = p1$; technically, this label indicates the value of variable p when the corresponding transition occurred. We call this the *transactional view* of an execution because we consider only the write and read operations of the process along with the commit and load continuation operations. All conditions and all other events are deleted form the execution. However, we keep the arcs between the remaining events: If there is a directed path of arcs of a particular type between two event in the original execution, then there is an arc of the corresponding type between these events in the transactional view.

Figure 29 shows the transactional view of another execution, which involves two processes $p1$ and $p2$ and two objects $o1$ and $o2$. Note that the computation of $p1$ crashes (after the first read event) and is restarted later on. Though the transition corresponding to the crash is masked in the transactional view of an execution, we know that $p1$ must have crashed after the read event and was restarted afterwards because of the `LdCnt(p1)` event. Note that there is no path of program causality from the first read event of $p1$ to the subsequent `LdCnt(p1)` event; there is only a normal path of causality.

Figure 29 can be considered as the execution of process `example` from Sect. 2.1, in the context of some other process $p2$. Object $o1$ corresponds to variable `x` and object $o2$ corresponds to variable `y`. Note, however, that there are 5 events corresponding to the execution of $p1$ due to the restart of $p1$ after a crash. Moreover, there is a commit event for process $p1$ that might have been inserted by a compiler at the end of the process.

4.1.3 Events, program causality, and data causality

Basically, an abstract view of an execution consists of a set of *events*, where each event corresponds to the occurrence of some operation. The corresponding operation is represented by a label. Moreover, there is a partial order (also called *causality*) which represents the order in which these events occur in this particular execution. Note that this causality is only a partial order; events which are not ordered by it are said to be concurrent.

In our version of executions, we identify two particular causalities: *program causality* and *data causality*. Program causality is graphically represented by arrows with a white arrow head. This causality represents the order between events that is due to the control flow of the executed sequential process (cf. Sect. 2.3.1–2.3.3). Each path of program causality corresponds to a computation of a sequential process as introduced in Sect. 2.1. Thus program causality allows us to identify the computations of the sequential processes within

the execution of the protocol. Note, however, that program causality is not a total order on the events of the same process! The reason is that, due to faults, a process may restart from an earlier continuation (cf. Fig. 29 and the discussion at the end of Sect. 4.1.2). But, we assume that the read and write events on each path of program causality in an execution are a legal computation of the process (cf. Sect. 4.3).

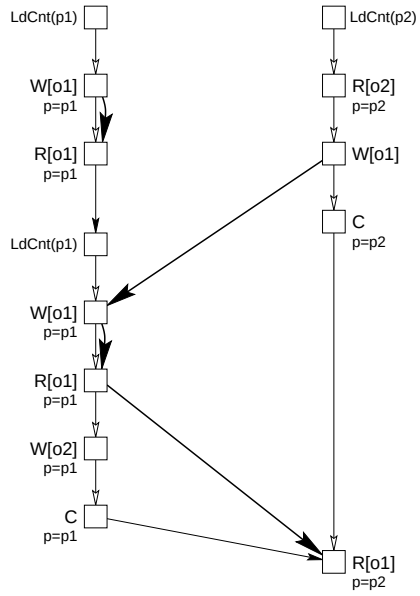


Figure 29: Transactional view of another execution

Data causality is graphically represented by bold-faced arrows. This causality represents the flow of values between certain write and read events. Though we do not represent values explicitly in our Petri net model, data causality reflects this information. We assume (cf. Sect. 4.3) that, for each read event, there is a uniquely defined path of data causality[8] to this event. The sequence of write events on this path determines the value returned by a read event. Thus, we choose this sequence as a representative of the value returned by the read event (see [21] for a more detailed discussion). For a read event e, we call this sequence of write events the *write sequence* of e and denote it by $ws(e)$.

In an execution, an event labeled by R[o1] corresponds to the occurrence of a read operation on object $o1$; we call this event a *read event* or—if the accessed object matters— a *read event on object* $o1$. Likewise, we call an event labeled by W[o1] a *write event* or a *write event on object o1*. An event that is either a write event or a read event is called an *access event*.

An event labeled by C is called a *commit event*. All events that can be invoked by the parallel program (or rather its sequential processes) are called *program events*; so a program event is either an access events or a commit event. Later on, we will also refer to events that correspond to the occurrence of some other transition t of the protocol; we call this a t *event* for short.

In the proof, we will refer to the conditions immediately preceding an event. The set of all conditions immediately preceding an event e is called the *preset* of e. Likewise, we call the immediate successor conditions of an event e the *postset* of e. The union of both sets is called the *context* of the event.

4.1.4 Program's view

Now, we introduce an even more abstract view of executions: the *program's view*. It consists of all program events (i.e. all access and commit events) of the execution along with the paths of program causality and the paths of data causality between these events. We drop all other events and causalities. Figure 30 shows the

[8]Whenever we refer to a path of some causality, we mean a directed path of this causality; so a path always follows the direction of the causality.

program's view of the execution represented in Fig. 29. Note that we dropped the causality from event $e2$ to

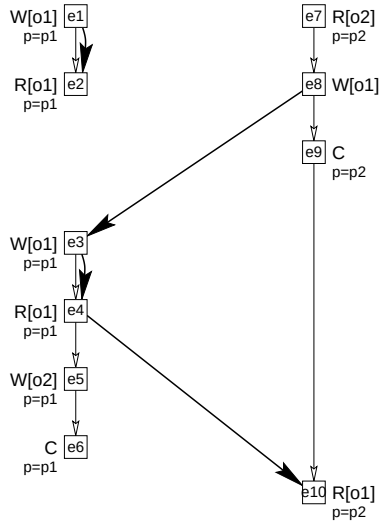


Figure 30: Programs view of the other execution

event $e3$ because it is neither a program causality nor a data causality. Likewise, we dropped the causality from event $e6$ to event $e10$.

The program's view of an execution provides all information necessary to specify the correct execution of a parallel program: Program causality reflects the computations of the sequential processes with its read and write operations on objects and its commit operations. Data causality reflects the propagation of values between write and read events. In particular, the write sequence $ws(e)$ associated with a read event e represents the value returned by this read event.

4.2 Specification

Now, we are prepared to formalize the informal understanding of correctness of the protocol in terms of the executions of the Petri net model. Informally, the execution of the sequential processes should appear as if no faults occurred and as being executed on a classical shared memory.

In terms of the program's view of the execution, this means that, for each process, we can identify a path of program causality that corresponds to a correct computation of this process. Moreover, the access events on these paths appear to be executed on a conventional shared memory.

We start by identifying such a path for each process by the help of commit events: An access event is *committed* if there exists a path of program causality from this event to some commit event. In the execution from Fig. 30, events $e3$, $e4$, $e5$, $e7$, and $e8$ are committed; events $e1$, $e2$, and $e10$, however, are not committed.

Our first requirement is that, for each process p, the set of committed access events of process p forms an *initialized path*[9] of process $p1$, where a set E of events forms an initialized path of process p if the following conditions hold:

1. For two events $e_1, e_2 \in E$ there exist a path of program causality either from e_1 to e_2 or from e_2 to e_1 (all events are totally ordered by program causality).

2. For an event $e \in E$ and any access event e' with a path of program causality from e' to e, we have $e' \in E$ (E is prefix closed with respect to program causality).

We will argue in Sect. 4.3 that any initialized path of a process p corresponds to a correct computation of process p.

[9]Remember that we consider directed paths only.

In the execution of Fig. 30, the set of committed access events of process $p1$ is $\{e3, e4, e5\}$, and the set of committed access events of process $p2$ is $\{e7, e8\}$. Both sets form an initialized path of the corresponding process.

Given these paths, we must make sure that the access events on these paths appear to be executed on a conventional shared memory with no interference of the other (non-committed) events. This is captured by the notion of *sequential consistency* [25]—slightly adapted to the situation of ignoring non-committed write events (see [21] for more details).

We require that the set of all committed access events is sequentially consistent. A set E of access events of a given execution is *sequentially consistent* if there exists a linear arrangement of E (i.e. a sequence in which each event of E occurs exactly once) such that:

1. the linear arrangement respects program causality; i.e. if two events are ordered by program causality, then these two events occur in the same order in the linear arrangement.

2. the value returned by the read event in the execution is the same as the value returned by the read event when the events are executed in the order of the linear arrangement on a conventional memory. In terms of our execution model this can be expressed as follows: Let e be a read event on object o. By $\widehat{ws}(e)$ we denote the sequence of write events on the same object which precede e in the linear arrangement. Then, the above requirement reads: For each read event e of X holds: $ws(e) = \widehat{ws}(e)$.

For example, the linear arrangement $e7, e8, e3, e4, e5$ is sequentially consistent, which covers exactly the committed events of the execution from Fig. 30. Note that ignoring non-committed events is implicitly covered by this definition: If an event e' does not belong to set E, it does not occur in the corresponding linear arrangement. Therefore, for no read event $e \in E$, the event e' occurs in $\widehat{ws}(e)$; because of the requirement $ws(e) = \widehat{ws}(e)$, we know that event e' does not occur in $ws(e)$. By definition of $ws(e)$, there is no data causality from e' to e in the execution.

To sum up, *an execution is correct* if,

A. for each process p, the set of its committed access events of p form an initialized path of p and

B. the set of all committed access events is sequentially consistent in the execution.

That's it—as far as safety is concerned. As already mentioned, this definition is based on some assumptions which will be discussed in Sect. 4.3 below.

The above requirement specifies a safety property, which says that a bad thing never happens [24, 2] (e.g. it is a bad thing to return a wrong value to a read operation). Thus, the specification could be easily implemented by never executing a commit event or even simpler by never executing any operation. Therefore, we need an additional liveness requirement, which says that eventually something good will happen (e.g. a good thing would be to successfully execute a commit event). In this paper, however, we concentrate on the specification of the safety property. The reasons are the following:

1. Liveness cannot be guaranteed at all without further assumptions on the fault model. If crashes occur frequently, it could happen that the protocol never recovers from a crash before the next crash. One way out of this would be to require that eventually no faults occur any more.

   This still leaves the problem of different processes conspiring against each other by not releasing locks. One way out of this would be to have a fairness assumption on acquiring and releasing locks. Remember that our protocol was allowed to insert commit points and thus to release locks.

   Then, we could apply classical techniques of temporal logic for proving this liveness.

2. The focus of this paper is on the consistency and the correct interplay of loading and saving continuations and objects. This requires a new way of thinking and new specification and proof techniques. Therefore, we concentrate on these new techniques and their applicability in this paper.

4.3 Assumptions

As already indicated in Sect. 4.2, the above specification is based on some assumptions.

First, we assumed that each initialized path of each process p (and thus by the requirement the set of committed events of that process) are a legal computation of this process. This assumption can be split into two parts:

1. We assume that in the execution each initialized path of a process p starts at the continuation of process p that is initially stored in the database.

   This assumption will be proved for the Petri net model by a simple structural argument in Sect. 8.4 (Property 12.2 & 4).

2. We assume that, whenever a continuation is loaded from the database and started as a process, the sequence of operations invoked by this process is a legal computation of the corresponding continuation.

   Moreover, we assume that, if a continuation of a process is saved at some point of the computation and later on loaded and started, then it resumes the computation at exactly the point where the continuation was saved.

   Note that this assumption cannot be proved within our Petri net model. We did not introduce a programming language for sequential processes along with a formal semantics and runtime system executing this program. Implementing a sequential programming language is a different issue and beyond the scope of this paper. In fact, the protocol works for any implementation of a sequential programming language that allows to save and restart continuations and meets the above requirements.

In combination, both assumptions guarantee that each initialized path of a process p starts as process p and keeps behaving as process p (even if stored as a continuation and restarted later on).

Note that the second assumption cannot be met if the program's semantics makes any real-time claims. For example, let us consider a situation where the next statement is a write operation and the program's semantics assumes it to be executed within $100ms$. Then saving the continuation to the database at this point and loading and restarting it–let's say a minute later–will violate the programs semantics. Thus, the second requirement excludes programs with real-time semantics. But, the second requirement does not exclude programs with a semantics depending on time (or timers).

There are several reasons why we make this an assumption rather than making it a part of the model:

1. This way, we need not deal with the semantics of the sequential processes at all. The semantics of sequential processes is a orthogonal (and complex) issue which requires different techniques. A formalization would not provide any insight into the protocol and would rather blur the proof of the protocol.

2. This way, we verify the protocol in a more generic way since we prove it independently of a particular programming language. In fact, it works with any programming language having any semantics for which the above assumptions are met.

Another assumption concerns the correct implementation of data causality. As stated before, we do not represent values in our Petri net model explicitly. Rather, we represent propagation of values between write and read events by data causality. Our interpretation is that a read event e returns the value which is the outcome of the sequence of write events $ws(e)$ preceding this read event with respect to data causality. This assumption can also be split into two assumptions:

1. We assume that the path of data causality to event e is unique. This is necessary for $ws(e)$ to be well-defined.

   This can (and will) be proved for the Petri net model by a simple structural property.

2. We assume that on each path of data causality (which corresponds to the paths on which data are propagated) the value of the object is faithfully propagated. On this path, only a write operation changes the value—according to its semantics. All other operations do not change the value[10].

   Again, this requirement cannot be proved within our Petri net model. It remains an assumption on the implementation since our model does not represent values at all.

There are several reasons for not including values in the Petri net model and to use data causality instead:

1. We need not bother about the domain from which the value should be chosen (we can leave it to the programs semantics to assign a domain to a particular object).

2. We need not even fix a particular version of a write operations. For example, a write could also be an atomic increment operation or even a 'multiply by two and than add one'. Moreover, we can allow write operations which only change a part of an object and do not touch another part (see [21] for a more detailed discussion). We allow any version of write operations which can be implemented such that the above requirement is satisfied.

In addition, we have some technical requirements on executions, which will be proven also in Sect. 8.4: We require that there is a path of data causality between access events on the same object only, and we require that there is a path of program causality between events of the same process only (Property 11). Moreover, we require that each path of data causality and each path of program causality originates in some initial condition of the execution (Property 12).

[10]Note that we do not require a particular representation of the value. Thus, there could be different representations of the same value on different physical devices.

5 Refinement

In the previous section, we have specified the correct behavior—i.e. the correct executions—of our protocol. Now, we are going to verify that each execution of the Petri net model meets this specification. Since the proof is quite long, we structure it in the following way: First, we will refine the specification in several steps. Each refinement resembles a design decision chosen in the protocol. For each refinement step, we will give a proof of correctness, meaning that each execution which meets the refined specification also meets the original specification. These proofs are independent of the protocol. At the end of the refinement process, we are left with a bunch of requirements, which will be proven for the Petri net model.

We start with general refinement steps in Sect. 5, which apply to many protocols and do not assume a particular operational implementation. In Sect. 6, we continue the refinement process with refinement steps that are driven by our particular protocol. In Sect. 7, we will verify the remaining requirements for the Petri net model.

5.1 Directly committed events

Both requirements, A and B, of the specification presented in Sect. 4.2 refer to the set of committed access events of an execution. Remember that an access event $e1$ is committed if there is a path of program causality from $e1$ to some commit event $e2$. Figure 31(a) shows a graphical representation of this definition. An event labeled by X denotes a read or write event to some object. The arc of program causality between event $e1$ and $e2$ denotes a path of program causality between these events.

Note that, according to this definition, the corresponding commit event $e2$ could be a commit event that occurs after an intermediate crash and after a restart of the process by a LdCnt event as shown in Fig. 31(b). In that case, we call $e2$ an *indirect commit event* of $e1$. The problem with indirect commit events is that the

(a) A committed event $e1$

(b) $e2$ is an indirect commit event of $e1$

Figure 31: Idea: Direct commit events

protocol is not aware of access events that happened before the restart of a process by a LdCnt event—in particular, an indirect commit event is not aware of a write event and cannot take responsibility for updating the value of the write event to the database. Therefore, it is difficult to verify requirements A and B directly for the protocol.

In order to tackle this difficulty, we proceed as follows: We introduce the concept of *directly committed events* and restrict requirements A and B to directly committed events. These restricted requirements will be called A' and B'. In order to guarantee the original specification, we add another requirement C, which guarantees that each committed event is a directly committed event.

We start with the definition of directly committed events: An access event $e1$ is a *directly committed event* if there exists a path of program causality from $e1$ to some commit event $e2$ on which no LdCnt event occurs. A graphical representation of this definition is shown in Fig. 32. Intuitively, a direct commit event occurs before the next crash of the process.

With this definition, the following three Requirements A', B', and C imply the specification from Sect. 4.2. Note that A' and B' rephrase requirement A and B for directly committed events only. In combination

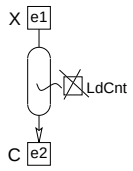


Figure 32: Graphical notation for a directly committed event $e1$

with Requirement C, this implies A and B.

***A':* *Directly committed events form an initialized path*:**
For each process $p1$, the set of all its directly committed events form an initialized path.

***B':* *Sequential consistency of directly committed events*:**
The set of all directly committed events is sequentially consistent.

***C :* *Committed implies directly committed*:**
Each committed event is a directly committed event.

5.2 Some notations: Situations

In the following sections, we will further refine requirements *A'*, *B'*, and *C*. In the presentation of these requirements, we will make use of some graphical notation, which was informally used in the previous section already. For example, Fig. 31(a) characterizes an execution with a committed access event $e1$. Figure 32 shows the additional requirement for directly committed events (there is no LdCnt event on the path of program causality from $e1$ to $e2$). Note that these figures apply to executions with more than two events since we only represent those events of an execution that are relevant for a particular purpose. We call those figures *situations*. An execution *meets* such a situation if we can identify the events of the situation in the execution and the events satisfy the constraints expressed in this situation. This way, a situation denotes a set of executions.

In the following, we will use implications between situations for representing requirements. For example, Fig. 33 represents Requirement C: For each access event $e1$ with a path of program causality to some commit event $e2$, there exists a path of program causality to some commit event $e3$ on which no LdCnt event occurs. Note that the event $e1$ on the left hand side and event $e1$ on the right hand side of the implication denote the same event. The commit events $e2$ and $e3$, however, may be different events because of the different names. But $e2$ and $e3$ could also be the same event (if $e2$ also satisfies the constraints imposed on $e3$). If we want two events to be different, we explicitly state this condition in the situation (e.g. by a label $e2 \neq e3$).

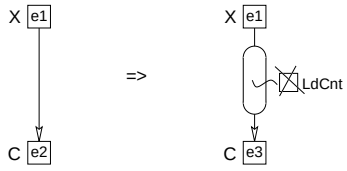


Figure 33: Graphical notation for Requirement C

5.3 Refining the initialized path property: A'

Next, we refine property *A'*, which we call the *initialized path property* for short. This property can be split into two requirements:

***A.1: Program causality between committed events*:** For each two directly committed events $e1$ and $e2$ that belong to the same process, there exists a path of program causality from $e1$ to $e2$ or from $e2$ to $e1$.

***A.2: No preceding uncommitted events*:** If there exists a path of program causality from an access event $e1$ to a directly committed access event $e2$ then event $e1$ is also a directly committed event.

Figures 34 and 35 show the graphical representations of these requirements in terms of implications between situations.

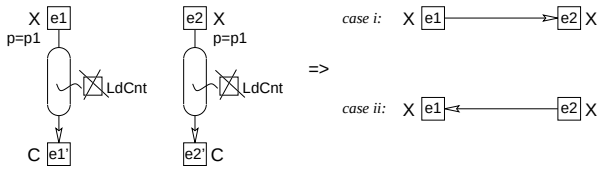


Figure 34: Requirement *A.1*: Program causality between committed events

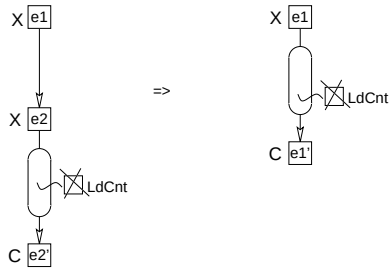


Figure 35: Requirement *A.2*: No preceding uncommitted events

Obviously, requirements *A.1* and *A.2* in combination guarantee the initialized path property *A'*. Requirement *A.1* guarantees that all directly committed events of the same process are totally ordered. Therefore, there exists a root anchored path that contains all directly committed events of this process. Requirement *A.2* guarantees that each access events on this path is directly committed because each event preceding a directly committed event is also a committed event.

5.4 Refining sequential consistency: B'

Next, we refine sequential consistency of the directly committed events (requirement *B'*). In the protocol, we use locks in order to guarantee exclusive access to each object. A part of this is captured by the following requirement. In this requirement, we use the concept of conflicting events: Two events are said to be *in conflict* (or *conflicting*) if both access the same object, both are directly committed events, and at least one of them is a write event.

***B.1: Conflicting events are ordered*:**
Every two conflicting events are ordered by causality in one way or the other.

This requirement is graphically represented in Fig. 36. The two different ways of arranging the conflicting events are graphically represented by two different cases on the right hand side of the implication. The two cases are distinguished by two labels *case i* and *case ii*.

A causal order between a write event $e1$ and a read event $e2$ on the same object, however, does not automatically imply that there is also a data causality from $e1$ to $e2$. For example, $e2$ could access an outdated copy. Therefore, we explicitly require this implication:

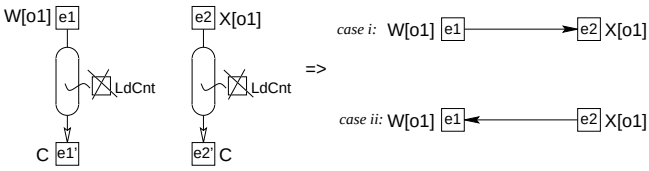


Figure 36: Requirement *B.1*: Conflicting events are ordered

***B.2: Causal order implies data causality*:** Let $e1$ be a directly committed write event on some object $o1$, and let $e2$ be a directly committed read or write event on the same object $o1$. If $e1$ happens causally before $e2$, then we also have a path of data causality from $e1$ to $e2$.

This requirement is graphically represented in Fig. 37.

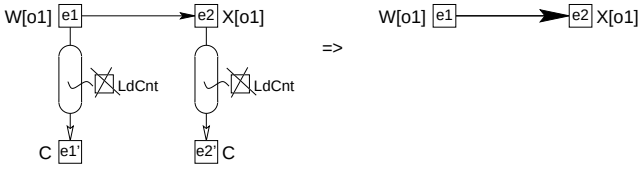


Figure 37: Requirement *B.2*: Causal order implies data causality

A last requirement guarantees that a directly committed event never gets a value from an event that is not directly committed. For convenience, we formalize the contraposition:

***B.3: Data causality respects commit events*:** If $e1$ is a write event with a path of data causality to some directly committed access event $e2$, then $e1$ is a directly committed event.

This requirement is graphically represented in Fig. 38.

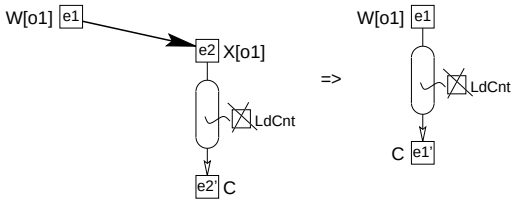


Figure 38: Requirement *B.3*: Data causality respects commit events

In combination, requirements *B1*, *B2*, and *B3* imply sequential consistency (requirement *B'*). Let us explain why. At first, we fix a linear arrangement of all directly committed events: an arbitrary linearization of the partial order on the committed events. By definition, this linear arrangement respects program causality (because program causality is a part of the partial order). So, it remains to show that $ws(e) = \widehat{ws}(e)$ for each committed read event e. Remember that $\widehat{ws}(e)$ depends on the chosen linearization. Let us consider a read event e on object $o1$. By *B.1*, we know that the partial order of the execution totally orders all committed write events on object $o1$ and the read event e. Let $e_1\, e_2 \ldots e_n\, e\, e_{n+1} \ldots$ be the sequence of these events. Since the linear arrangement was a linearization of this order, we know $\widehat{ws}(e) = e_1\, e_2 \ldots e_n$. By *B.2*, we know that there is a path of data causality from write event e_i to write event e_{i+1} for each i with $1 \leq i < n$. Also by *B.2*, we know that there is a path of data causality from e_n to e. It remains to show that there are no other write events on the path of data causality towards event e. By assumption on our execution model (Prop. 11.1), there is no path of data causality from a write event on a different object $o2 \neq o1$ to event e. So it is sufficient to consider the write events on object $o1$: For each write event e_i with $i \geq n+1$, there cannot be is no path of data causality from e_i to e because, by definition of the sequence,

there is a causality from e to e_i—a contradiction to anti-symmetry of partial orders. So, the only additional events on the path of data causality towards e could be write events on object $o1$ that are not directly committed. This, however, is excluded by requirement *B.3*. Thus, we have $\widehat{ws}(e) = ws(e)$.

Altogether, we know that requirements *B.1*, *B.2*, and *B.3* imply requirement *B'*.

5.5 Summary and discussion

In the previous sections, we have refined requirements *A'* and *B'*. For the moment, we do not further refine requirement C because it will turn out that requirement C will be implied by two requirements that occur during the further refinement *A.2* (see Sect. 6.11). Altogether, it remains to show that requirements *A.1*, *A.2*, *B.1*, *B.2*, *B.3*, and C are met by the Petri net model. These requirements will be refined further in the next section.

However, it is worthwhile to notice the similarities between the requirement *A.1* and the requirements *B.1* and *B.2*. Both establish a total order on events which are somehow related: *A.1* guarantees that, for two directly committed events of the same process, there is a program causality between these events. *B.1* and *B.2* guarantee that, for two directly committed write events on the same object, there is a data causality between these events. Indeed, if there were no read events, we could replace the two requirements *B.1* and *B.2* by a single requirement which resembles *A.1*. Since we do not require a data causality from a read event to a write event, we need the combination of *B.1* and *B.2*. Even more evident, requirement *A.2* and requirement *B.3* have the same structure.

These similarities are not by chance. Whenever the process executes a read or write event, the process's continuation is modified. Therefore, from the continuation's point of view, each read or write event can be considered a 'write event on the continuation'! A 'write event on the continuation' basically needs the same treatment as a write event on some other object—the only difference is that propagation of continuations is represented by program causality and not by data causality.

This analogy will also pervade the following refinement steps and even the final verification steps. The arguments in the case for continuations will be slightly simpler than the arguments in the analog case for objects—since a continuation can be changed only by one process whereas an object can be changed by different processes.

6 Further refinement

In this section, we further refine the specification. We start by refining requirements *A.1* and *A.2*. Figure 48 on page 48 gives an overview of all refinement steps concerning requirement *A'*. In particular, there are references to the page numbers on which the corresponding requirements are defined. Therefore, we do not repeat the precise wording of the requirements which have been defined earlier. We give only a brief informal description for the requirement to be refined.

6.1 Refining A.2

We start by refining requirement *A.2* because this refinement step is straightforward—compared to the refinement of *A.1*. Moreover, the resulting requirements will not be refined further.

Requirement *A.2* (see p. 38) says that there is no program causality from any not directly committed event to a directly committed event. Let us consider two events that are related by program causality. The following requirement allows to distinguish between two cases:

***A.2.1: Two cases of program causality*:** If $e1$ is an access event with a program causality to some program event $e2$, then either

case i: we have a path of program causality from $e1$ to $e2$ on which an event $e3$ with a `cnt.p1` condition in its postset and a `LdCnt(p1)` event $e4$ occurs (cf. Fig. 39). Moreover, there is no `LdCnt` event on the path of program causality from $e1$ to $e3$.

case ii: we have a path of program causality from $e1$ to $e2$ on which no LdCnt event occurs.

Requirement *A.2.1* is graphically represented in Fig. 39, where the distinction between the two cases is indicated by corresponding labels. *Case i* corresponds to a restart of process $p1$ between events $e1$ and $e2$ due to a crash. In *case i*, the process does not crash between $e1$ and $e2$. Note that this requirement applies to all access resp. program events—not only to the committed ones. Note that we refer to the occurrence of a place in this situation. Remember that we call this occurrence a *condition*, and that a condition represent the occurrence of a particular token on a certain place in the Petri net model. In a situation, the interpretation of an arc between an event and a condition differs from the interpretation of an arc between two events: An arc between two events represents a path of the corresponding causality between these events in the underlying execution. The path may even have length 0; in that case, the events are identical. In contrast, an arc between a condition and an event in a situation means that the corresponding condition is in the preset of the event—i.e. there is exactly one arc from the condition to the event in an execution that meets this situation. Similarly, an execution meets a situation with an arc from an event to a condition only if the condition is in the postset of this event. Thus, an arc between a condition and an event (or vice versa) means that the condition occurs in the *immediate context* of the event. The condition denotes a token consumed or produced by this particular event.

Due to this interpretation, we introduced another condition $c2$ between condition $c1$ and $e4$ in the graphical representation of requirement *A.2.1*. If $c2$ was omitted, *case i* would require that $c1$ is also in the preset of event $e4$. Note that the two conditions $c1$ and $c2$ could be identical in the corresponding execution. An arc between two conditions is interpreted as a (possibly empty) path of the corresponding

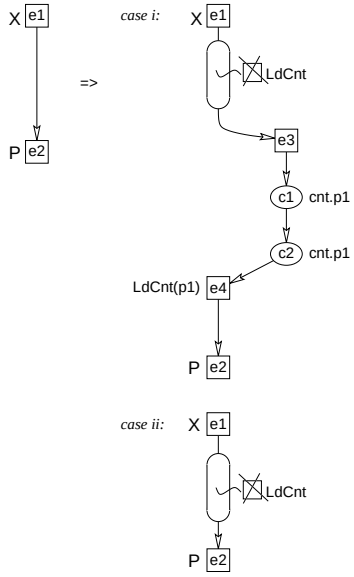


Figure 39: Requirement *A.2.1*: Two cases of program causality

causality between these conditions. Since arcs between the same kind of nodes do not occur in Petri nets and their executions, we will not be tempted to interpret these arcs as an immediate context. Altogether, our interpretation of arcs, captures the spirit of Petri nets: All places that are related by an arc to a transition form the context of this transition.

Let us consider requirement *A.1* again. In order to guarantee this requirement we must make sure that a read or write event is directly committed whenever its continuation is written to the database (cnt.p1). This is formalized by the following requirement:

***A.2.2: Written continuation implies commit*:** Let $e1$ be an access event, and let $e2$ be an event with an arc of program causality to some cnt.p1 condition c. Moreover, let there be a path of program

causality from $e1$ to $e2$ on which no LdCnt event occurs. Then, there is a path of program causality from $e1$ to some commit event $e3$ on which no LdCnt event occurs, and $e2$ happens causally after the commit event $e3$.

Requirement *A.2.2* is graphically represented in Fig. 40.

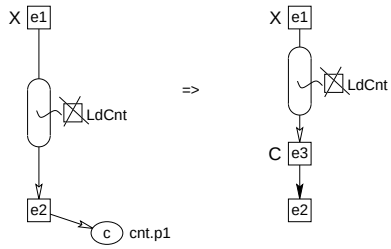


Figure 40: Requirement *A.2.2*: Written continuation implies commit

Requirement *A.2* is an immediate consequence of requirements *A.2.1* and *A.2.2*: Let us consider a situation in an execution that meets the left hand side of requirement *A.2*. In particular, we know that there is an access event $e1$ with a program causality to some directly committed program event (remember that an access event is a program event by definition). Now, we must show that in this situation $e1$ is a directly committed event. Obviously, the above situation meets the left hand side of *A.2.1* also. Therefore, we have one of the situations on the right hand side of *A.2.1*. In *case i*, the situation meets the left hand side of *A.2.2*, and therefore, we know that $e1$ is a directly committed event; finishing our proof in that case. In *case ii*, there is a path of program causality from $e1$ to $e2$ on which no LdCnt event occurs. Moreover, in the considered situation $e2$ is a directly committed event, which by definition, means that we have a path of program causality from $e2$ to some C event e' on which no LdCnt event occurs. Since paths of causality are transitive (and $e2$ was not a LdCnt event), we have a path of program causality from $e1$ to a commit event e' on which no LdCnt event occurs; by definition, $e1$ is a directly committed event.

The above argument may appear a bit excessive. From the graphical representation of the requirements, this implication is obvious. The purpose, of the detailed arguments above is to provide some feeling for the involved techniques and on the meaning of the graphical representations of the requirements—without giving a formal definition of their semantics. After some further examples of such arguments, we will start skipping such obvious arguments.

6.2 Refining A.1

Next, we refine requirement *A.1* (see p. 38). This requirement says that two directly committed access events of the same process are ordered by program causality in one way or the other. The refinement of this requirement resembles the idea already used in requirements *B.2* and *B.3*: First, we require that all access events of the same process are totally ordered (not necessarily by program causality); second, we require that, from a directly committed event that happens causally before any other access event of the same process, there is also a program causality to this other event.

***A.1.1: Order between access events of the same process*:** Let $e1$ and $e2$ be access events of the same process. Then, there is a path of causality from $e1$ to $e2$ or there is a path of causality from $e2$ to $e1$.

***A.1.2: Order implies program causality*:** Let $e1$ and $e2$ be access events of the same process, and let $e1$ be a directly committed event. If there is a path of causality from $e1$ to $e2$, then there is also a path of program causality from $e1$ to $e2$.

Requirement *A.1.1* is graphically represented in Fig. 41, and requirement *A.1.2* is graphically represented in Fig. 42.

In combination, requirements *A.1.1* and *A.1.2* imply requirement *A.1*. Again, the argument is simple: Let us consider a situation in an execution that meets the left hand side of requirement *A.1*; i.e. there are

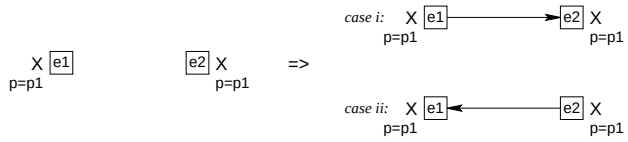


Figure 41: Requirement *A.1.1*: Order between access events of the same process

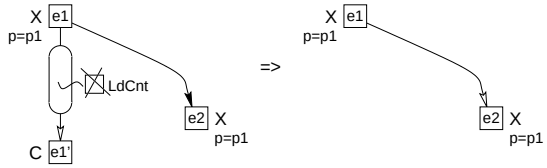


Figure 42: Requirement *A.1.2*: Order implies program causality

two directly committed events $e1$ and $e2$ that belong to the same process $p1$. We have to show that there is a path of program causality from $e1$ to $e2$ or vice versa. The situation meets also the left hand side of *A.1.1*. Therefore, we know that $e1$ or $e2$ are ordered in one way or the other. Without loss of generality, we assume that $e1$ happens causally before $e2$. By assumption, $e1$ is a directly committed event. Therefore, the left hand side of *A.1.2* meets this situation, which implies that there is a program causality from $e1$ to $e2$.

6.3 Refining A.1.2

Next, we further refine requirement *A.1.2* (see above) by distinguishing the following two cases: First, no process crash occurs between $e1$ and $e2$; second, a process crash between $e1$ and $e2$, and the process is restarted by a CheckIn(p1) and a LdCnt(p1) event.

***A.1.2.1: Two cases of event ordering*:** Let $e1$ and $e2$ be two access events which belong to the same process $p1$. Moreover, let there be a path of program causality from $e1$ to some commit event $e3$ on which no LdCnt event occurs. Then, either

case i: there is a path of program causality from $e1$ to $e2$ on which no LdCnt event occurs, or

case ii: there is a CheckIn(p1) event $e4$ and a LdCnt(p1) event $e3$ such that $e4$ occurs causally between $e1'$ and $e3$, and there is a path of program causality from the LdCnt(p1) event $e3$ to $e2$.

This requirement is graphically represented in Fig. 43. Remember that we did not yet prove this requirement for the Petri net model. For the moment being, we only claim that this requirement along with requirement *A.1.2.2*, which will be defined below, implies requirement *A.1.2*.

In order to guarantee *A.1.2*, we must make sure that in *case ii* of *A.1.2.1* there is also a path of program causality form $e1$ to $e2$. This is guaranteed by requirement *A.1.2.2*—via the LdCnt event.

***A.1.2.2: Restart after commit implies program causality*:** Let $e1$ be an access event of some process $p1$ with a direct commit event $e1'$. Moreover, let there be a CheckIn(p1) event $e4$ and a LdCnt(p1) event $e3$ such that $e4$ occurs causally after the commit event $e1'$, and $e3$ occurs causally after $e4$. Then, there is a path of program causality from $e1$ to $e3$.

Requirement *A.1.2.2* is graphically represented in Fig. 44.

Obviously, requirements *A.1.2.1* and *A.1.2.2* imply requirement *A.1.2*. Therefore, we start omitting detailed arguments at this point.

6.4 Refining A.1.2.2

At last, we further refine requirement *A.1.2.2*. First, we state some more details on the relation between the commit event $e1'$ and the LdCnt(p1) event $e3$ in the situation of the left hand side of requirement *A.1.2.2*.

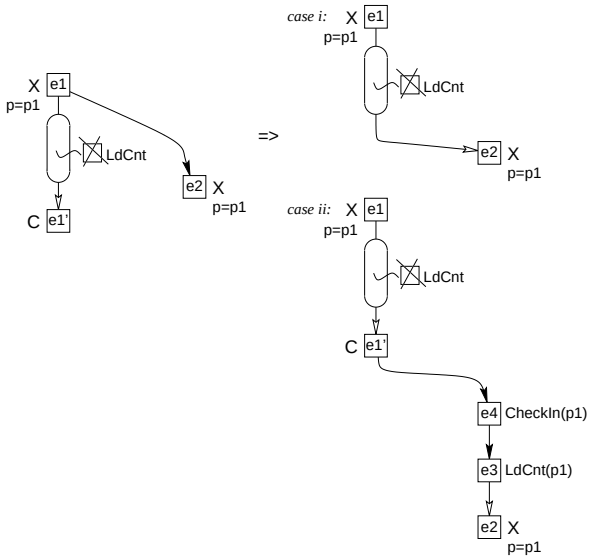


Figure 43: Requirement *A.1.2.1*: Two cases of event ordering

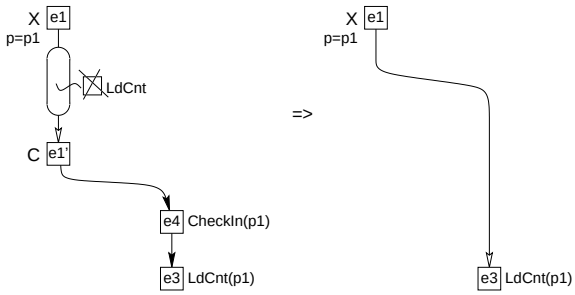


Figure 44: Requirement *A.1.2.2*: Restart after commit implies program causality

This will be captured by requirement *A.1.2.2.1*, which is graphically represented in Fig. 45. Since we

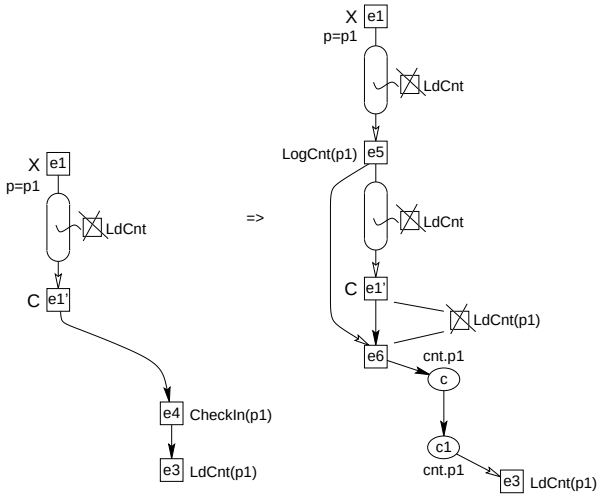


Figure 45: Requirement *A.1.2.2.1*: Order of writing and reading continuations to or from the database

introduce some a new notation in this graphical representation, we start with explaining this new notation. In Fig. 45 there is a crossed LdCnt(p1) event between $e1'$ and $e6$, where the position is graphically indicated by two lines. It indicates that no LdCnt(p1) event can occur 'between' $e1'$ and $e6$. Since 'between' is slightly ambiguous in the context of partial orders, we make the meaning of this more precise: Each LdCnt(p1) event either happens causally before $e1'$ or happens causally after $e6$. Note that this interpretation is different from the meaning of the crossed LdCnt event between $e1$ and $e1'$. This crossed event only claims that there is no LdCnt event on the particular path of program causality from $e1$ to $e1'$; in principle, there could be another path of program causality from $e1$ to $e1'$ on which a LdCnt event occurs. Indeed, there are paths between $e1$ and $e1'$ on which a LdCnt event occurs in some executions of the protocol; but these paths are not program causality and the event is not a LdCnt(p1) event.

Altogether, *A.1.2.2.1* says that, before the LdCnt(p1) event $e3$ can load some continuation from the database (indicated by the corresponding condition $c1$), the continuation of $e1$ is written to the database. Moreover, there are no LdCnt(p1) events between the commit event $e1'$ and the write continuation event $e6$. Note that the write continuation event is not required to be successful since the arc from $e6$ is not a program causality—but it could be. Moreover, requirement *A.1.2.2.1* adds some context: the event that writes the continuation to the log file (event $e5$). Note *A.1.2.2.1* does not tell whether $e6$ uses the log file or the current state of the process for writing the continuation to the database—both cases are possible. The later case occurs if no crash occurs; the first case occurs if the process crashes before writing its continuation to the database.

The situation shown in Fig. 45 is quite complex, and a textual description of the corresponding requirement is quite clumsy. Since the graphical notation has a precise semantics, we will not give a textual description for this and most of the following requirements. For the sake of completeness, we will give a number and a reference to the corresponding figure within the text:

A.1.2.2.1 See Fig. 45

Requirement *A.1.2.2.1* does not guarantee that there is a program causality from $e1$ to $e3$. In order to guarantee this (as required by *A.1.2.2*), we introduce another requirement: For a possible intermediate event $e7$ that writes the continuation $c1$, there either exists a path of program causality from $e1$ to $e7$ or event $e7$ gets its continuation from another read continuation event $e8$ that happens causally after c (cf. Fig. 46).

A.1.2.2.2 See Fig. 46.

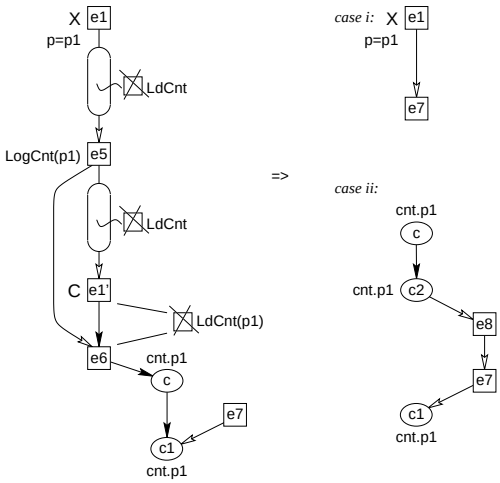


Figure 46: Requirement *A.1.2.2.2*: Intermediate events don't skip $e1$ and c

Now let us see why requirements *A.1.2.2.1* and *A.1.2.2.2* imply requirement *A.1.2.2*: We consider a situation of an execution that meets the left hand side of requirement *A.1.2.2*. We must show that there is a path of program causality from $e1$ to $e3$. Since the left hand sides of *A.1.2.2* and *A.1.2.2.1* are identical, *A.1.2.2.1* meets also the above situation. Therefore, we have the situation on the right hand side of *A.1.2.2.1*. This situation is shown in Fig. 47 again. We will refer to this situation by $(*)$ in the following.

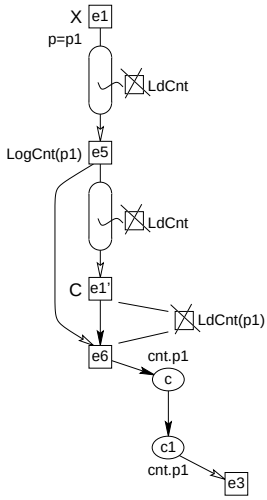


Figure 47: Situation $(*)$

Now, it remains to show that there is a path of program causality from event $e1$ to condition $c1$. We will proof this by repeated application of requirement *A.1.2.2.2*. To this end, we consider the[11] event $e7$ which immediately precedes $c1$. We know that such an event exists since $c1$ is not an initial condition ($e6$ is a preceding event). Moreover, by our assumptions on executions (Prop. 12.2), there is a path of program causality from $e7$ to $c1$ because there is a program causality from $c1$ to $e3$. Therefore, the left hand side of requirement *A.1.2.2.2* meets this situation. Applying the requirement, we have one of the following two situations:

[11] Note that an event that immediately precedes a condition is uniquely defined if it exists at all.

case i: There is a path of program causality from $e1$ to $e7$. By transitivity, we also have a path of program causality from $e1$ via $e7$ to $c1$.

case ii: There is another event $e8$ between c and $c1$ that reads a continuation $c2$, and there is a path of program causality from $c2$ to $c1$.

This situation also meets the situation $(*)$ when $c2$ takes the role of $c1$ and $e8$ takes the role of $e3$. So we can apply the same arguments as above, where we substitute $c2$ for $c1$ and $e8$ for $e3$.

By recursive application of the above arguments, we either end up with a condition c_n for which there is a program causality from $e1$ to c_n and a program causality from c_n via all intermediate c_j to $c1$ (if *case i* applies at this stage) or we end up with an infinite chain of conditions $c1, c2, c3, \ldots$ that all occur between c and $c1$. The second case, however, is impossible because an infinite chain of conditions between two conditions is not possible in an execution.

A careful look at the above proof reveals that we did not use the fact that LdCnt(p1) events are excluded between the commit event and the event writing the continuation ($e2$ and $e6$). Similarly, we did not use the details about the context of event $e1$ which are added by requirement *A.1.2.2.1*. Indeed, we do not need this requirement for proving the refinement step; these details will be necessary, however, for proving requirement *A.1.2.2.2* for the Petri net model.

6.5 Overview: Requirement A'

Before refining requirement *B'* in a similar way to requirement *A*, let us briefly sum up all remaining proof obligations. Figure 48 represents all refinement steps for requirement *A'* as a tree. Each node shows the number of the requirement along with the page number of its definition. Since we have proven the correctness of each refinement step, we are now left with the requirements at the leafs of the refinement tree. These are: *A.1.1*, *A.1.2.1*, *A.1.2.2.1*, *A.1.2.2.2*, *A.2.1*, and *A.2.2*.

6.6 Refining B.3

In the following, we will refine requirements *B.1*, *B.2*, and *B.3*. An overview on all refinement steps can be found in Fig. 59 on page 55. In analogy to the refinement of requirement *A*, we start by refining *B.3*.

Requirement *B.3* (see p. 39) is refined to requirements *B.3.1*, *B.3.2*, and *B.3.3*, which are represented in Fig. 49, Fig. 50, and Fig. 51 respectively.

We do not go into all details of these requirements because they are similar to the requirements *A.2.1* and *A.2.2*. The new part here is the inscription of a path by cp.(p1,o1,wr) in requirement *B.3.1*. This means that each condition on this path of data causality is inscribed by cp.(p1,o1,wr). This basically reflects the fact, that in *case ii* the data causality from $e1$ to $e2$ is via the same local copy of object $o1$ that was acquired as a write copy. Requirement *B.3.3* says that, in that case, both events are executed by the same process during the same execution phase—i.e. without an intermediate crash resp. LdCnt event. Note that the dot in the third component of cp.(p1,o1,.) indicates that we do not restrict the value of this component. Therefore, the copy can be a read or a write copy. Requirement *B.3.2* resembles requirement *A.3.2*: If a change is ever written to the database, the corresponding write operation is committed.

Altogether, requirements *B.3.1*, *B.3.2*, and *B.3.3* imply requirement *B.3*. The arguments are analog to the ones given in Sect. 6.1 for the refinement of requirement *A.2*: Let us consider a situation that meets the left hand side of requirement *B.3*; i.e. there is a data causality from a write event $e1$ to an access event $e2$, and $e2$ is a directly committed event. We must show that $e1$ is also a directly committed event. The considered situation meets also the left hand side of requirement *B.3.1*. Thus, we have one of the two cases on the right hand side of *B.3.1*. In *case i*, requirement *B.3.3* guarantees that there is a path of program causality from event $e1$ to $e1'$ on which no LdCnt event occurs; requirement *B.3.2* guarantees that there is a path of program causality from $e1'$ to some commit event on which no LdCnt event occurs. Altogether, we know that $e1$ is a directly committed event in *case i*. In *case ii*, requirement *B.3.3* guarantees that there is a path of program causality from $e1$ to $e2$ on which no LdCnt event occurs. Since, by assumption, $e2$ was a directly committed event, $e1$ is also a directly committed event.

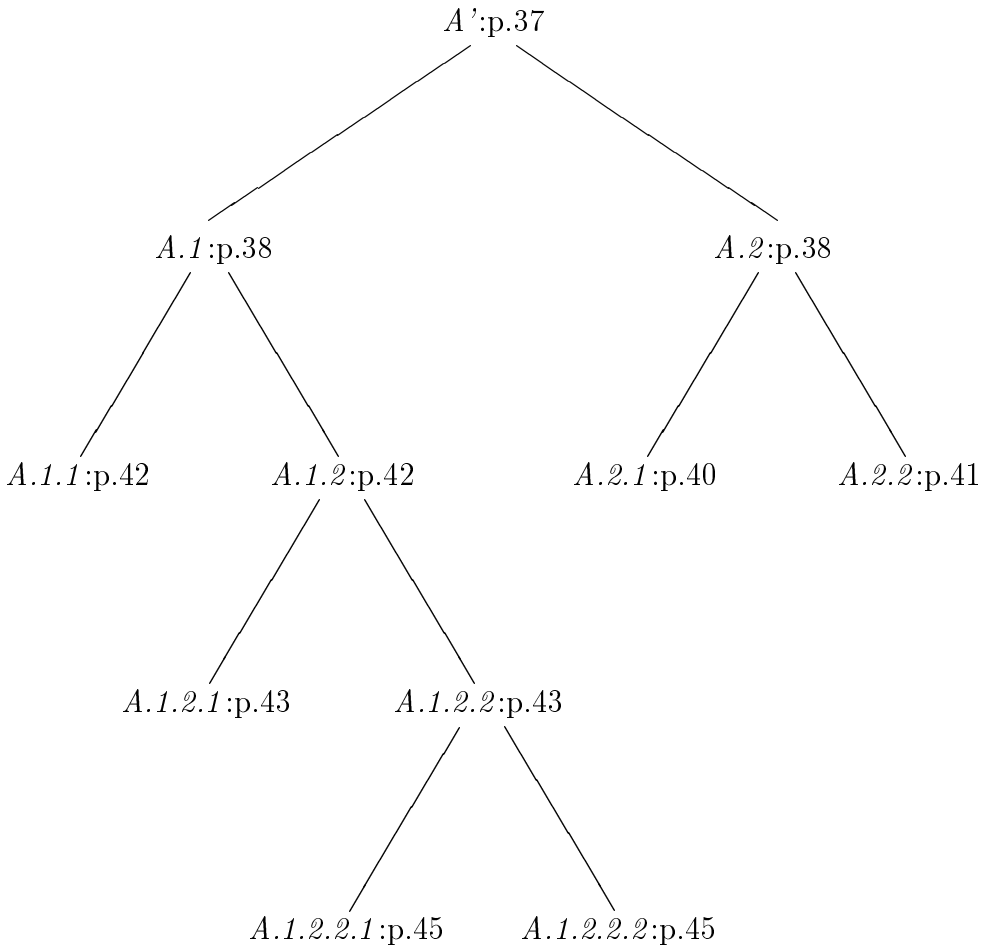


Figure 48: Overview on refinement steps of requirement *A'*

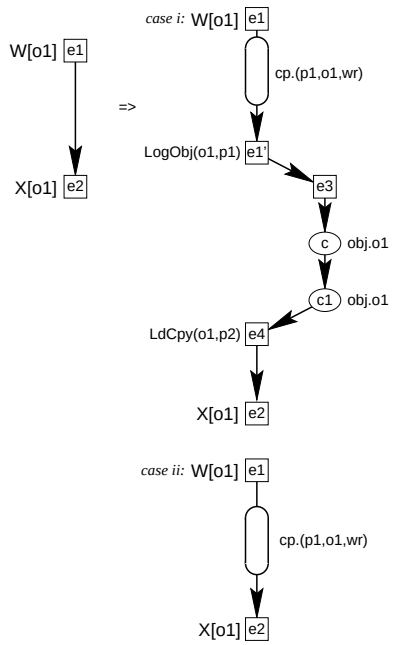


Figure 49: Requirement *B.3.1*: Two cases of data causality

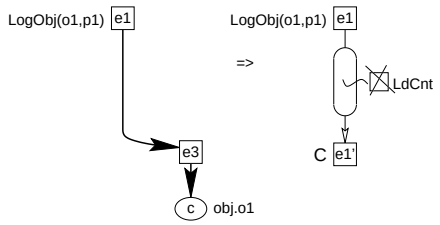


Figure 50: Requirement *B.3.2*: Updated object implies direct commit

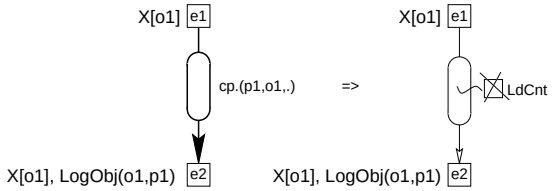


Figure 51: Requirement *B.3.3*: Access on same copy implies same execution phase

6.7 Refining B.1

Next, we refine requirement *B.1* (see p. 38) by the three requirements *B.1.1*, *B.1.2*, and *B.1.3*, which are graphically represented in Fig. 52 through Fig. 54. Requirements *B.1.1* gives the context in which a

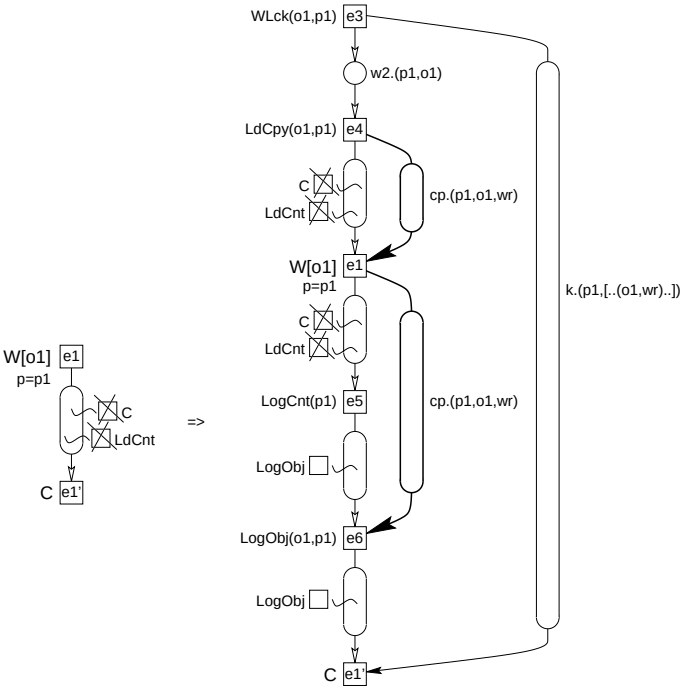


Figure 52: Requirement *B.1.1*: Context of a write event

directly committed write event occurs—starting from the lock operation up to the commit operation. In particular, the paths of data causality are precisely defined. Note that in the case of a write event the paths of data causality between the corresponding events only show cp.(p1,o1,wr) conditions. Moreover, the server knows process $p1$ from the lock event until the commit event occurs. This is indicated by the path labeled by k.(p1,[..(o1,wr)..], where the pair $(o1, wr)$ indicates that object $o1$ is exclusively locked for process $p1$ throughout.

Similarly, *B.1.2* gives the context of an access event (i.e. a read or a write event; thus *B.1.1* is a special case of *B.1.2*). Since an access event can be either a read or a write event, we do not have as much detailed information as in the case of a write event. But, we still know that the object is propagated from the LdCpy(o1,p1) event to the access event. Moreover, the object is locked at the server from the lock event until the access operation has happened. Note that the lock is not necessarily exclusive. We indicate this by the label k.(p1,[..(o1,.)..]). The label Lck(o1,p1) for event $e3$ indicates that $e3$ is a WLck(o1,p1) event or a RLck(o1,p1) event.

Requirement *B.1.3* formalizes that a write lock is exclusive: Let $e1$ and $e2$ be two lock events on the same object $o1$ by processes $p1$ and $p2$, where $e1$ is a write lock (WLck) and $e2$ is a read or a write lock (Lck). Moreover, let there be a path of k.(p1,[..(o1,wr)..]) conditions from $e1$ to some event $e3$, and let there be a path of k.(p1,[..(o1,.)..]) conditions from $e2$ to some event $e4$. Then, *B.1.3* requires that either $e3$ happens causally before $e2$ (*case i*), or $e4$ happens causally before $e1$ (*case ii*), or $e1$ and $e2$ coincide (*case iii*). Note that we have $p1 = p2$ in *case iii*.

For two conflicting events of two different processes, the three above requirements imply that these events are causally ordered in one way or the other as required by *B.1*. For two access events of the same process we have already required in *A.1.1* that they are ordered in one way or the other—even if the events are not conflicting.

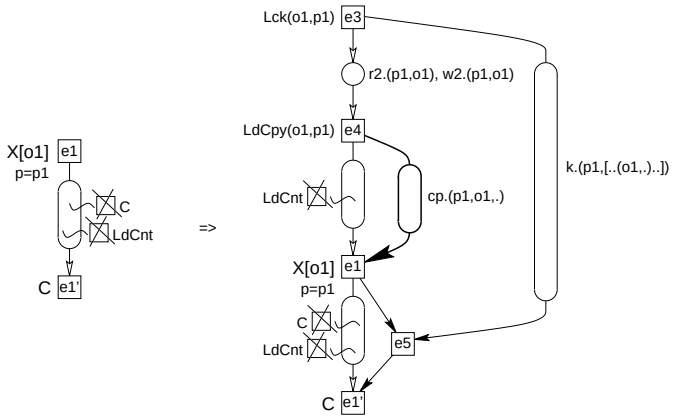


Figure 53: Requirement *B.1.2*: Context of an access event

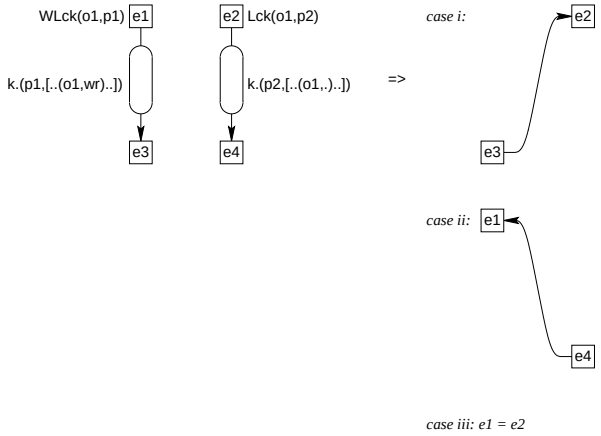


Figure 54: Requirement *B.1.3*: Write locks are exclusive

Altogether, requirements *B.1.1*, *B.1.2*, *B.1.3*, and *A.1.1* imply requirement *B.1*.

6.8 Refining B.2

Now, we refine requirement *B.2* (see p. 39) by distinguishing between two cases, which are then dealt with separately. This step is almost the same as the refinement of requirement *A.1.2* (see Sect. 6.3).

Let $e1$ be a write event on some object, and let $e2$ be some access event on the same object that occurs causally after $e1$. Then, requirement *B.2.1* distinguishes between two cases:

***case i*:** both events access the object via the same copy.

***case ii*:** $e2$ accesses a different copy, which is loaded by a LdCpy event $e5$ causally after a direct commit event for event $e1$ and after a Lck event.

In *case i*, requirement *B.2* is valid because of the path of data causality from $e1$ to $e2$. In *case ii*, requirement *B.2.2* guarantees that there is a path of data causality from $e3$ to $e5$. By transitivity, there is also a path of data causality from $e1$ via $e3$ and $e5$ to $e2$.

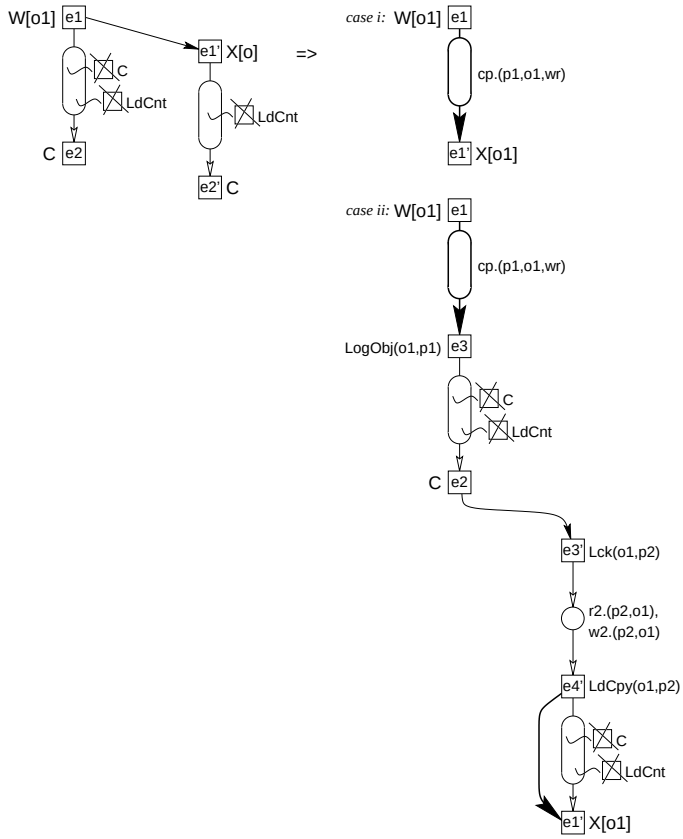


Figure 55: Requirement *B.2.1*: Two cases

6.9 Refining B.2.2

At last, we refine *B.2.2* (see p. 53), which resembles the refinement of *A.1.2.2*. Indeed, the arguments for correctness are the same as in Sect. 6.4. Here, we only have a more complex context for the write event. This context, however, is not needed for proving the correctness of the refinement step; it is only necessary (or at least helpful) for proving the resulting requirements for the Petri net model.

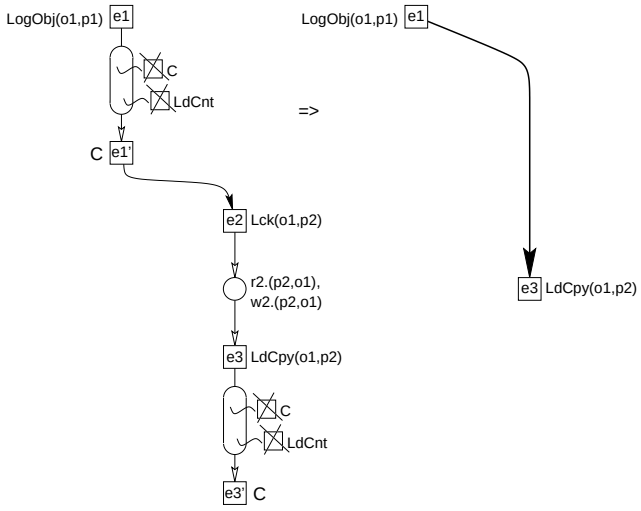


Figure 56: Requirement *B.2.2*: Correct global propagation

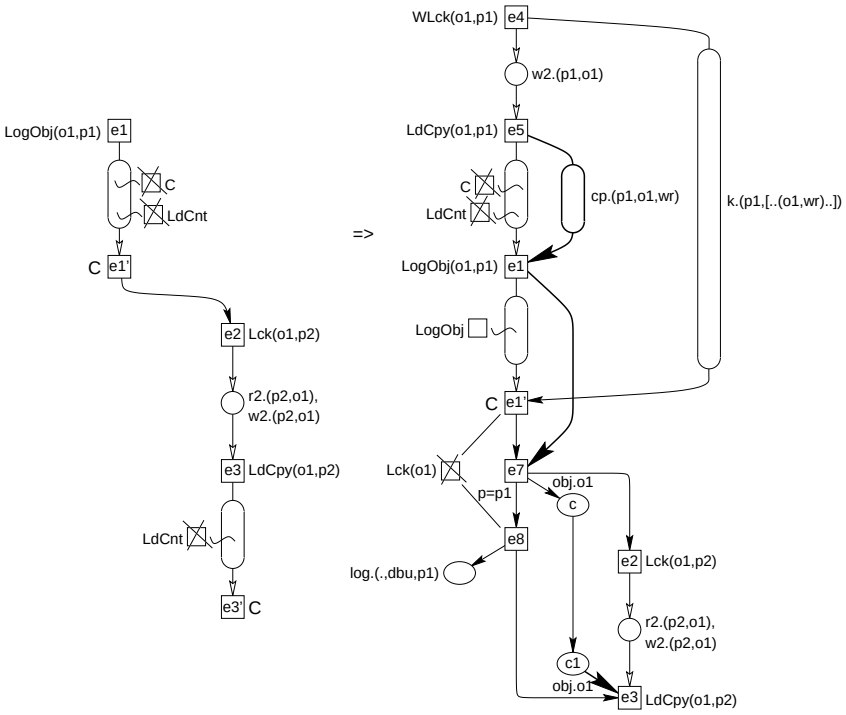


Figure 57: Requirement *B.2.2.1*: `LdCpy(o1)` after update of object and writing of update record to log.

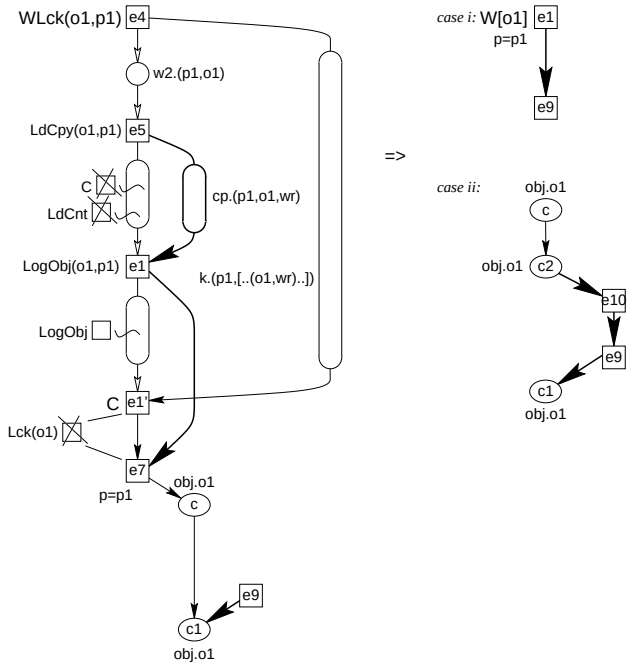


Figure 58: Requirement *B.2.2.2*: No skip of updates.

6.10 Overview: Requirement B

Figure 59 gives an overview on the refinement steps of requirement *B'*. The remaining proof obligations are *B.1.1*, *B.1.2*, *B.1.3*, *B.2.1*, *B.2.2.1*, *B.2.2.2*, *B.3.1*, *B.3.2*, and *B.3.3*.

Note that we used requirement *A.1.1* for proving the refinement of *B.1*—this obligation was already listed at the end of the refinement *A*. Therefore, we do not list it here again.

6.11 Refining C

At last, we refine requirement *C* (see p. 37), which states that each committed access event is also a directly committed event. However, we need no further proof obligations because requirement *C* is a consequence of requirement *A.2.1* and *A.2.2* (see p. 40 and p. 41). Let us consider a situation that meets the left hand side of requirement *C*: an access event $e1$ with a path of program causality to some commit event $e2$. Since a commit event is a program event, this situation meets also the left hand side of requirement *A.2.1*. Applying *A.2.1* gives us two cases: In *case ii*, $e2$ is a direct commit event for $e1$. In *case i*, the situation meets the left hand side of requirement *A.2.2*. Application of *A.2.2* shows that $e1$ is a directly committed event.

Altogether, refinement of requirement *C* does not add any further proof obligations.

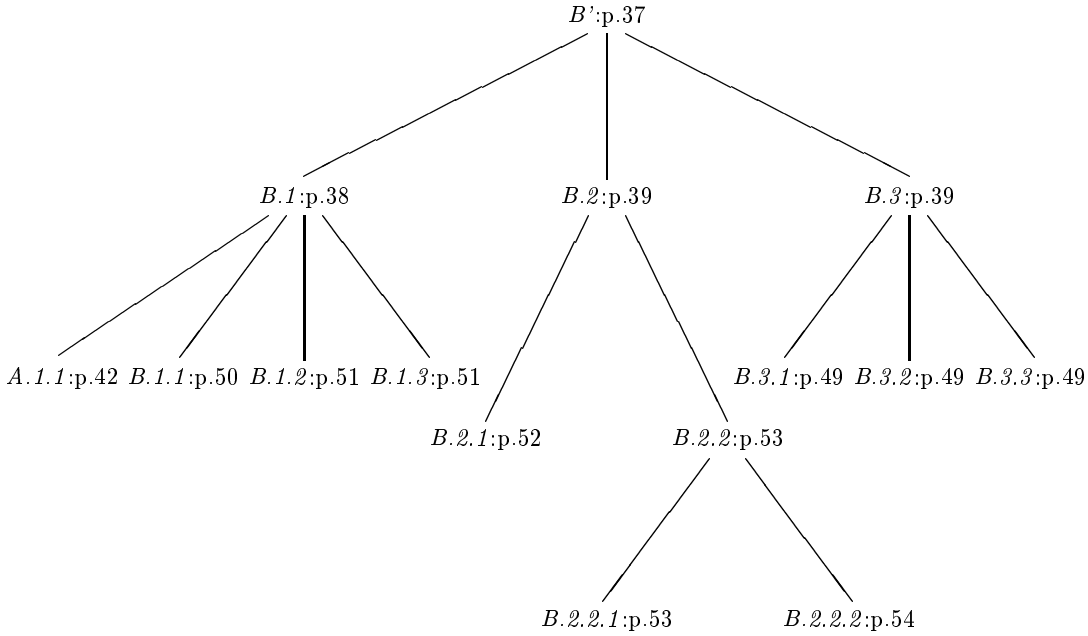


Figure 59: Overview on refinement steps of requirement B

7 Verification

In the previous sections, we have refined the specification such that each requirement is an implication between two situations. In the rest of this paper, we show that each execution of the Petri net model meets these requirements. This will be proven for each requirement separately.

The basic structure of each proof is the following: We consider an arbitrary execution of the Petri net model that meets the right-hand of the requirement. Then, we construct the context of the considered situation from the Petri model. For this construction, we use different kinds of automata, which extract the relevant behavior of the Petri net model: all paths between specific events, the chain of causes of some event, etc. Details will be explained below.

Up to know, this proof is hand-made. The used techniques, however, are such that they can be automated. For example, most automata used could be constructed fully automatically from the requirement and the Petri net model; and most verification steps could be supported by techniques from automated theorem proving. Unfortunately, these techniques are not yet implemented—a tool supporting this technique is an ongoing project.

For understanding the verification technique, it is sufficient to consider the proofs of the first few requirements—all other requirements follow the same arguments. Nevertheless, we do not skip the proof of the other requirements and give a proof for each requirement. Each reader can decide himself where to stop ...

7.1 A.2.1: Two cases of program causality between access events

In this section, we verify requirement *A.2.1* (see p. 40). This requirement says that a path of program causality from one access event $e1$ to another access event $e2$ is either via some condition cnt.p1 for some $p1$ or there is a path of program causality from $e1$ to $e2$ on which no LdCnt event occurs.

This can be proven by an automaton that represents all possible paths of program causality between two access events in our Petri net model, which is shown in Fig. 60. In this automaton, we represent each transition that has a program causality arc removing a token from some place and that has a program causality arc adding a token to some place. In order to reduce the graphical representation, we have only drawn one place a. Place a holds a token $p1$, if there is a token $p1$ on one of the places process, c1, c2, c3, c4, c5 or c6, or there is some token $(p1, o1)$ on one of the places r1, r2, w1, w2. A token $p1$ on place a basically represents an active process (Table 4 on page 104 list all abbreviations used in the proof). Note

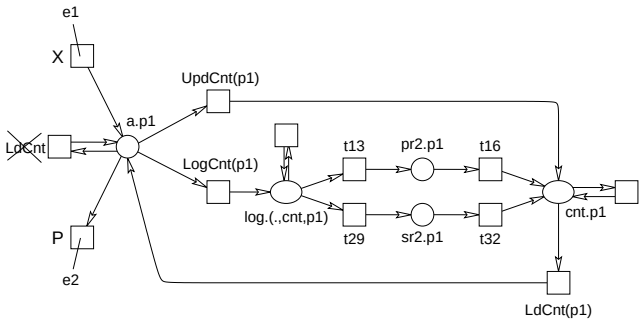


Figure 60: Automaton for program causality between two access events

that most transitions which remove a token $p1$ from one of these places also add one token $p1$ to one of these places. This is captured by the transition with a loop to place a. We do not really care which are these transitions except that it is not a LdCnt event—a LdCnt event does not remove a token $p1$ from one of the places corresponding to a (see below). This transition could for example be a UpdCnt(p1) event or a LogCnt(p1)—both remove a token $p1$ from one of the above places and add one token $p1$ to it. A closer look to the Petri net model of these operations on the server's side in Fig. 8 and Fig. 9 reveals that both transitions do not only have a program causality arc to one of the above places, but also have a program

causality arc to cnt and log. In an execution, this corresponds to a split of program causality. Since the automaton should represent all paths, we also need to include these two possibilities, which are represented by the corresponding transitions in the automaton. The LdCnt transition removes a token from cnt and also adds a token to it (via arcs of program causality); this is represented by the transition with a loop to cnt. The LdCnt transition, however, also adds a token to place a via a program causality arc; so we again have two instances of LdCnt(p). The transitions that correspond to the considered events $e1$ and $e2$ are represented separately and labeled accordingly. Altogether, the automaton from Fig. 60 represents all paths of the Petri net model between events $e1$ and $e2$. Thus, each path of program causality from an access event $e1$ to some program event $e2$ in an execution of the Petri net model is represented by a path from $e1$ to $e2$ in the automaton from Fig. 60.

Now, let us have look at the possible paths. Clearly, there are paths from $e1$ to $e2$ on which no LdCnt event occurs (*case ii* of *A.2.1*). If there is a LdCnt event on a path from $e1$ to $e2$, this path is via a cnt.p1 condition. Moreover, there is no LdCnt before the first occurrence of a condition cnt.p1 on that path (*case i* of *A.2.1*). This proves requirement *A.2.1* for all executions of our Petri net model..

7.2 A.2.2: Written continuations are committed

Now, we verify requirement *A.2.2* (see p. 41): If a continuation of some access event $e1$ is ever written to the database (a cnt.p1 condition $c1$) for some $p1$, then $e1$ is a committed event.

Again, we will use automata for proving this property. But, we will not explain the automaton in full detail anymore. The proof requirement *A.2.2*, however, is a little bit more involved because there are three different ways of writing a continuation to the database: by an update event during the execution of the commit operation (UpdCnt), by an update event during the redo of an individual process (t16), or by an update event during the restart of the server (t32).

These different cases are represented by the automaton in Fig. 61. In this automaton, a' stands for a subset of the places of a used in the previous automaton; we exclude places c2 and c3 because we want to have a closer look to what is going on during the commit operation at this stage. Again, a definition of a' can be found in Table 4 on page 104. Moreover, we do omit the LdCnt operation in this automaton because the assumption of A.2.2 only considers paths on which the LdCnt event does not occur.

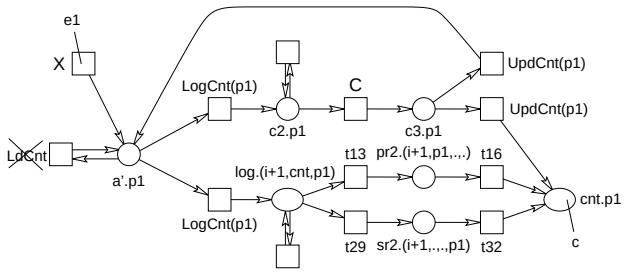


Figure 61: Automaton for program causality between an access event and a cnt.p1 condition

From the automaton, we can identify the following three cases:

case i: The path is via a UpdCnt(p1) event (executed during the commit operation); in that case there is a commit event on this path. Since there are no intermediate LdCnt events, $e1$ is a directly committed event.

case ii: The path is via a LogCnt(p1) event and the continuation is written by an occurrence of transition t16 (i.e. it is updated from the log during the redo of a process).

case iii: The path is via a LogCnt(p1) event and the continuation is written by an occurrence of transition t23 (i.e. it is updated from the log during the restart of the server).

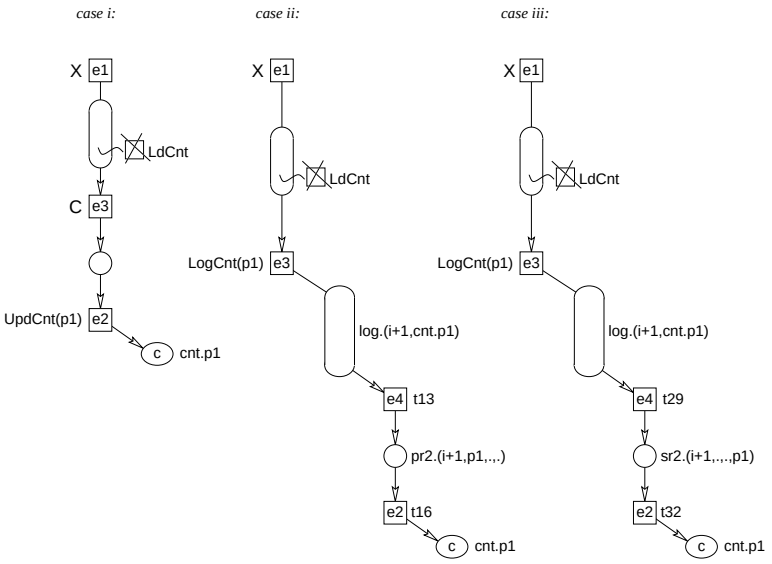


Figure 62: *A.2.2*: Three possible cases

These cases are graphically represented in Fig. 62.

In *case i*, the proof is already finished. In *cases ii* and *iii*, we will show below that there is a commit event $e5$ for process $p1$ that happens after $e3$ and that there are no LogCnt(p1) events in between as shown in Fig. 63. Then by Property 1, which will be proven on page 105, we know that there is a path of program causality from $e3$ to $e5$ on which no LdCnt events occur. By transitivity, we know that $e1$ is a directly committed event.

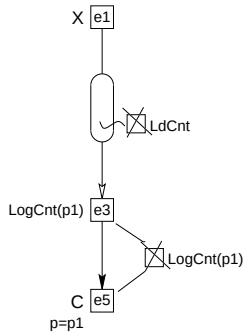


Figure 63: There exist a commit event for $p1$ without an intermediate LogCnt(p1) event.

For the moment, it remains to show that, in *cases ii* and *iii*, we can find a commit event $e5$ as shown in Fig. 63. Informally, the argument is the following: If a continuation of process $p1$ is updated in the database during a backwards scan, then there must have been a commit record for process $p1$ encountered before and there are no log continuation records in between. Clearly, the commit record must have been written by a *commit event* $e5$ for process $p1$. Since, the log is scanned backwards, the log must have happened causally after the LogCnt(p1) event $e3$; since all intermediate log records were different from a log continuation record, we know that there is no LogCnt(p1) between $e3$ and $e5$. In the following, we will give formalize these arguments in terms of the Petri net model for *case ii* and *case iii* separately.

Case ii: What is necessary for transition t16 to occur with $p = p1$? First of all, we know that the token on pr2 must be $(i+1, p1, true, p1)$ for some i since t16 can only occur in that case (see Petri net model in

Fig. 12 on p. 15); remember that *true* in the third component of this tuple means that a commit record for $p1$ was encountered during the redo procedure for $p1$. This token is produced by transition t13; in order to produce this token, transition t13 needs a token $(i+1, p1, true)$ on place pr. Since this place is also initially unmarked, there must be an event that produced this token, which needs some other tokens in order to occur—and so on.

Figure 64 shows an automaton that represents the *chain of causes* for transition t13: all possible paths that could produce a token $(i+1, p1, true)$ on place pr: It could be either produced by transitions t10, t12, t14, t19, t22, or by t20. These transitions, in turn, need a token on places pr1, pr2, pr4, or pr5, which are characterized in more detail in the automaton. By the way, these transitions decrement the value of i, which is indicated by a label i:= i-1 at the corresponding transition. Note that we introduce a counter i in the automaton, in order to keep the representation of the automaton finite. We will use variable i for representing a counter in all our automata; incrementation of the counter and decrementation of the counter will be indicated by labeling the corresponding transition.

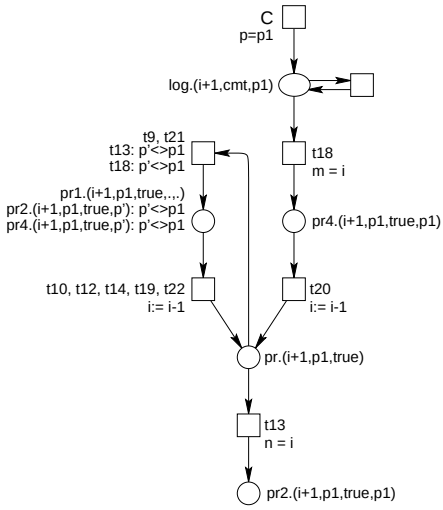


Figure 64: Automaton showing the chain of causes for t13.

What do we know from this automaton? Since all the places occurring in this automaton are initially unmarked, we know that there must once have been a commit event for process $p1$—otherwise the chain of causes would be infinite, which is impossible in our execution model. Let us call this commit event $e5$. Next, we show that the commit event $e5$ happens causally after the LogCnt(p1) event $e3$ (cf. Fig. 62). This is where the counter comes in. We now fix a particular value for the tuple $(i+1, cmt, p1)$ on place log which was produced by commit event $e5$—remember that i was used as a counter variable. Let us assume that $e5$ produced log.(m+1,cmt,p1). Let us assume also that, in the end (i.e. last occurrence of t13), the counter is n. In the graphical representation of the automaton, these assumptions are indicated by labels m = i and n = i at the corresponding transitions. Since all transitions of the automaton only decrease the value of i, we know $m \geq n$. We even know $m < n$ since, on each path from C to t13, there is at least one occurrence of a transition that decrements the counter: transition t20. Moreover, we know that both events, $e3$ and $e5$ access the log counter and m and n is the value of the log counter when these events occur. From the Petri net model, we know that, the value of the log counter is only increased. Therefore, we know that $e1$ happens causally before $e5$. This will be formalized by Property 5 on page 108. By applying this property to the above situation, we know that $e3$ happens causally before $e5$.

So it remains to prove that there is no LogCnt(p1) event between $e3$ and $e5$. To this end, let us consider the automaton again: On each path from C to t13 the events that read a log record with sequence number $i+1$ (events: t18, t9, t13, and t18 and t21) and the events that decrement the counter i strictly alternate. Thus, for each k with $n < k < m$, we have a log record different from $(k+1, cnt, p1)$: t18 reads a log record

$(k+1, cmt, p')$, t9 reads a log record $(k+1, o, p')$, t21 reads a log record $(k+1, dbu, p')$, t13, and t18 read a log record $(k+1, cmt, p')$. Transition t13 reads a log record $(k+1, cnt, p')$; by the condition $p' <> p1$ imposed on transition t13 in the automaton, however, we know that this is no log continuation record for $p1$. Now, let us assume that we have some LogCnt(p1) event that occurs between $e3$ and $e5$. By Property 5, we know $n < k < m$. Thus, the LogCnt(p1) event writes a log record (k+1,cnt,p1). In combination, we have two different log records with the same sequence number $k+1$. This, however, is a contradiction to Property 6 on page 109, which says that there is at most one log record for each sequence number. Thus, there is no LogCnt(p1) event between $e3$ and $e5$, which finishes the proof of *A.2.2* for *case ii*.

Case iii: The proof of *case iii* is similar to the proof of *case ii*. Again, we characterize the chain of causes of event t32 and t29. We know that transition t32 can only occur if we have a token $(i+1, ., [..p1..], p1)$ on place sr2 for some i. Where $[..p1..]$ represents a multiset in which at least process $p1$ occurs—remember that the third component of the tuple represents the multiset of processes for which a valid commit event has been encountered during the scan. The automaton in Fig. 65 traces back the path to the commit event which caused $p1$ to be added to this set.

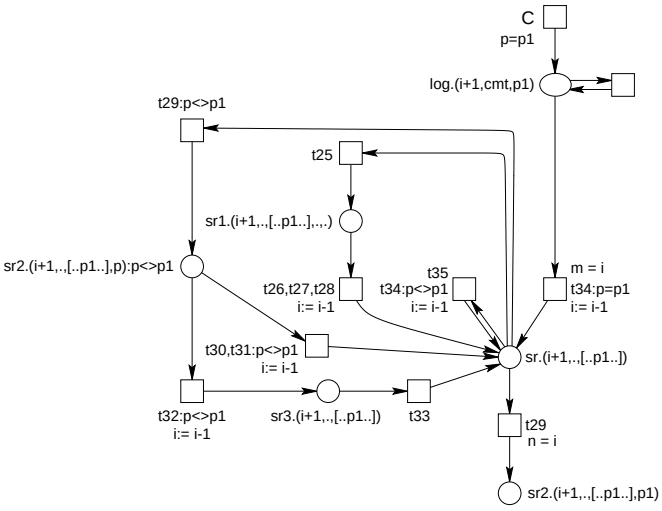


Figure 65: Automaton showing the chain of causes for t29.

The rest of the argument is identical to the one in *case ii*. Since all places of this automaton are initially unmarked in the Petri net model, we know that there is a chain of causes for t29 which once originated in a commit event $e5$ for process $p1$. This commit event writes a log record with a sequence number $m+1$ with $m > n$ because there are only transitions that decrease counter i and there is at least on such transition (t34). Therefore, $e5$ happens causally after $e3$ (by Property 5). Moreover, we know that on the path from $e5$ to $e4$ a log record $(k+1, ..)$ that is different from $(k+1, cnt, p1)$ is read for each k with $n < k < m$—due to the alternation of the events that decrease i and the events that read a log record. Note that t34 and t35 reads a log record and then decreases i. Again, a careful analysis of all events in this automaton shows that no log record which is read on this path is a log continuation record for process $p1$: The only transition that reads a continuation record is t29; this transition, however, is restricted to $p \neq p1$ in the automaton. By Property 6 and Property 5, we know that there is no intermediate LogCnt(p1) event between $e3$ and $e5$—otherwise there are two different log records with the same sequence number.

7.3 A.1.1: Access events of the same process are ordered

Now, we prove requirement *A.1.1* (see p. 42), which says that two access event which are executed by the same process are ordered in one way or the other.

Let us consider two access events $e1$ and $e2$ which are executed by the same process. Figure 66 shows this situation, were we add the condition in its preset and in its postset according to the preset and postset in the Petri net model. Since an access event is either a read event or a write event, we know that the condition in its preset is r1.(p1,o1) for a read event on some object $o1$ or w1.(p1,o1) for a write event on some object $o1$. The condition in the postset is p.p1—in both cases. However, we don't need this detailed information for proving the property, we have indicated that it is one of the places active (abbreviated a) which was already used before and is defined in Table 4. Remember that the conditions are in the immediate context of the events; in particular $c1$ and $c3$ are two different conditions and $c2$ and $c4$ are two different conditions if $e1$ and $e2$ are different events. Conditions $c2$ and $c3$ as well as conditions $c4$ and $c1$, however, could be the same.

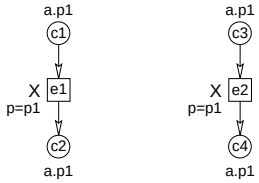


Figure 66: Two access events of the same process and their context

Now, let us consider the situation shown in Fig. 66. Let us assume that $e1$ and $e2$ are different events—otherwise there is nothing to prove. There is an invariant (invariant (1) on page 105) which says that at no time there is more than one token $p1$ on place a (or rather on the collection of places represented by a). Thus, we know that there is a causality from $c2$ to $c3$ or from $c3$ to $c4$—otherwise two different conditions with a label a.p1 are concurrent, which violates the invariant. Thus, each two different access events of the same process are causally ordered in one way or the other.

7.4 A.1.2.1: Two cases of ordered access event of the same process

Now, we prove requirement *A.1.2.1* (see p. 43). We have to prove the following: Let us consider two access events $e1$ and $e2$ of the same process of some process $p1$, let there be a path of program causality from $e1$ to some commit event $e1'$ on which no LdCnt event occurs, and let $e2$ happen causally after $e1$. We have to show that there either exists a path of program causality from $e1$ to $e2$ on which no LdCnt event occurs (*case i*) or that there exists a CheckIn(p1) event $e4$ followed by a LdCnt(p1) event $e3$ such there is a path of program causality from $e3$ to $e2$, and $e4$ occurs causally after $e1'$.

The proof is in three steps: First, we show that, in the above situation, there is a path of program causality from $e1$ to the corresponding commit event $e1'$ on which only conditions a.p1 occur (i.e. a token $p1$ or $(p1, o1)$ on places process, r1, r2, w1, w2, c1, c2, c3, c4, c5, c6; see Table 4 on page 4). This will be proven in Property 2 on page 105. Note that $e1$ is an access event of process $p1$ and has a a.p1 condition in its postset, and $e1'$ is a commit event of process $p1$ and has a a.p1 condition in its preset—thus, Property 2 applies to this situation. Intuitively, this property says that the process has been running without a crash if there is no intermediate LdCnt event.

Second, there exists a LdCnt(p1) event $e3$ with a path of program causality to $e2$ on which only conditions a.p1 occur. This is proven in Property 4 on page 108. Intuitively, this property says that each program event there must have been a LdCnt event and the process has not yet crashed since. Moreover, $e3$ has a s.p1 condition c in its preset; condition c in turn has a CheckIn(p1) event in its preset (see Petri net model on page 10).

Figure 67 shows the situation after these two steps.

Third, we exploit the invariant $\mathsf{a}[p1] + \mathsf{cr}[p1] + \mathsf{s}[p1] = 1$ (invariant (1), see Table 5 on page 105), which basically says that each process is exactly in one of the following states: it is running (at exactly) one point in the protocol, it has crashed, or it has just checked in and is about to load its continuation from the database. From this invariant, we know that no two conditions of the two paths from $e1$ to $e1'$ and from $e4$ to $e2$ shown in Fig. 67 can be concurrent. Therefore, all conditions on this paths are linearly ordered. Since

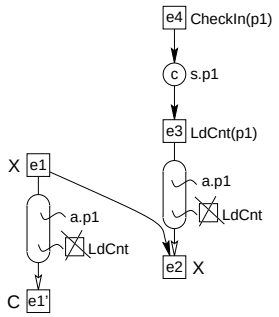


Figure 67: Situation in the proof of *A.2.1.1*

$e1$, $e1'$, $e4$, and $e2$ have these conditions in their immediate context, these events are linearly ordered too. The two paths could happen one after the other or both paths could overlap.

We consider all possible linearizations of the events on this two paths:

case i Event $e1$ happens on the path of program causality from $e4$ to $e2$ (due to the invariant, it must be on this path). Note that $e1$ and $e4$ do not coincide because CheckIn events are and access events correspond to different transitions; thus, $e1$ happens on the path from $e3$ to $e2$. In this case, we have a path of program causality from $e1$ to $e2$ on which no LdCnt events occur by assumption. Thus, we have the situation of *case i* of *A.1.2.1* (see above).

case ii Event $e4$ happens causally after event $e1'$. This is *case ii* of *A.1.2.1* (see above).

case iii Event $e4$ happens causally between $e1$ and $e1'$. According to invariant (1) above, condition s.p1 occurs on the path from $e1$ to $e1'$. This, however, is impossible; we know that only conditions a.p1 do occur on this path; s is not contained in the places denoted by a (see above). Therefore, *case iii* is impossible.

Altogether, we have shown that in a situation which matches the left hand side of requirement *A.1.2.1* we have the two cases on the right hand side of requirement *A.1.2.1*.

7.5 A.1.2.2.1: Correct order of update and loading from database

In this section, we prove requirement *A.1.2.2.1*, which basically says that a LdCnt operation and its preceding CheckIn operation that happens causally after some commit event is deferred until the corresponding continuation is written to the database (see p. 45). This is the most involved proof within of all properties in the refinement tree of requirement A. The reason is that many different protocols can interfere when writing a continuation to the database: After the commit point of a process, the process updates its continuation in the database by issuing an UpdCnt(p1) event. The process, however, could crash before the process has completed this update phase by writing its DBu record. In that case, the server will start the redo procedure for this process. The action taken by the server during the redo of the process depend on when the process crashed. Even worse, the server could crash during the redo procedure of the process. In that case, the update is completed by the restart procedure for the server—again, the concrete actions taken depend on how far the update procedure of the process itself and how far the update procedure during the redo of process could proceed before the two crashes occurred. All these cases are captured in the proof below. But, we do not distinguish the cases according to our understanding of the protocol in the proof; we distinguish the cases according to the structure of the Petri net model. This way, we are sure not to overlook one of the many cases.

Step 1, Setting up the scenario: We start with a situation that meets the left hand side of requirement *A.1.2.2.1*. This situation is shown in Fig. 68. We have added already some context to this situation: By a simple automaton for program causality, we know that there is a LogCnt(p1) event $e5$ on the path from

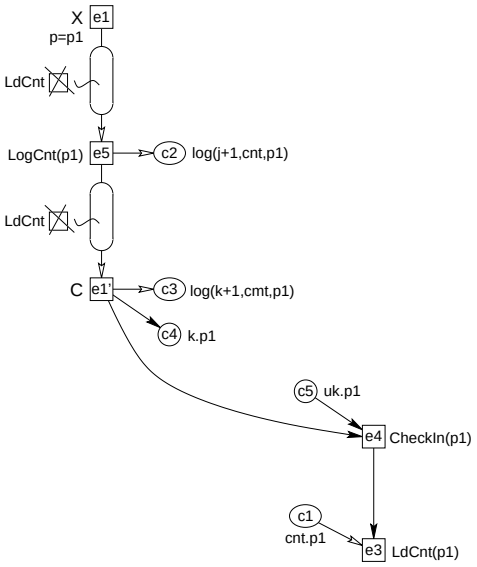


Figure 68: A situation that meets the left hand side of *A.1.2.2.1*

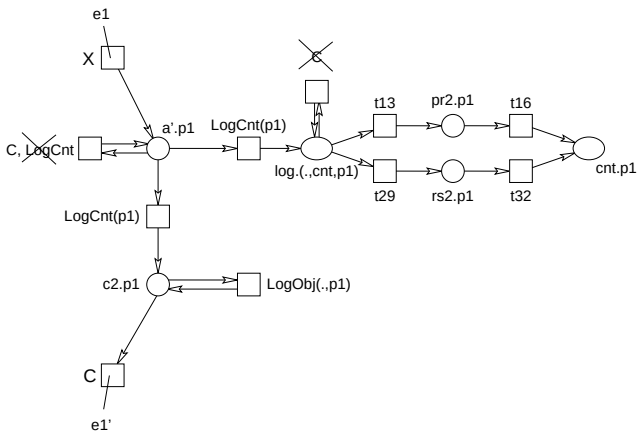


Figure 69: Automaton which proves existence of `LogCnt(p1)` event

the access event $e1$ to the commit event $e1'$. This automaton is shown in Fig. 69. For each event in this situation, we have added some of conditions in the immediate context of events $e1'$, $e5$, $e3$, and $e4$. This context can be picked up from the Petri net model itself.

By assumption, we know that event $e1'$ happens causally before event $e4$. Moreover, we have the invariant

$$\mathsf{k}[p1] + \mathsf{uk}[p1] + \mathsf{rel}[p1] + \mathsf{pri}[p1] + \mathsf{red}[p1] + \mathsf{sri} = 1$$

(invariant (2), see Table 5 on page 105) which says that, concerning a process $p1$, the server can be exactly in one of the following states: Process $p1$ is known to the server, it is not known to the server (but the server is not in one of the redo, start or crashed phase), it is in the redo phase of process $p1$, it is in the restart phase, or it is crashed. From this invariant, we know that conditions $c4$ and $c5$ must be ordered by causality in one way or the other. Since the commit event $e1'$ happens causally before the CheckIn event $e4$, we know that $c4$ is causally before $c5$.

Step 2, from know to unknown: By invariant (2), we know that there is a path of causality from $c4$ to $c5$ on which only conditions corresponding to one of the places of invariant (2) occur. In particular, there is a condition $c6$ that is the first occurrence of a uk.p1 condition on this path as shown in Fig. 70. By assumption

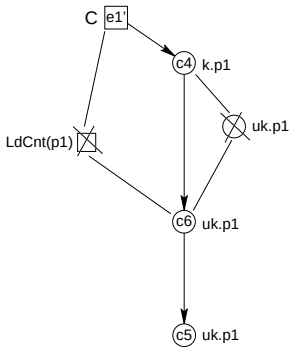


Figure 70: A path corresponding to invariant $\mathsf{k}[p1] + \mathsf{uk}[p1] + \mathsf{rel}[p1] + \mathsf{pri}[p1] + \mathsf{red}[p1] + \mathsf{sri} = 1$

and invariant (2), we know that there is no occurrence of a condition uk.p1 between conditions $c4$ and $c6$. Next, we show that there is no LdCnt(p1) event between $e1'$ and $c6$. Let us consider an arbitrary LdCnt(p1) event e: By invariant (1) and the conditions a.p1 in the context of events e and $e1'$, we know that e occurs causally before $e1'$ or causally after $e1'$. In the first case, e does not occur between $e1'$ and $c6$. So, let us consider the case that e occurs causally after $e1'$ in more detail. The automaton in Fig. 71 shows a chain of causes for the LdCnt(p1) event e: There must be a CheckIn(p1) event e' before, with a uk.p1 condition in its preset. By invariant (1), e' occurs after the commit event $e1'$ because, otherwise, s.p1 is concurrent to a a.p1 condition in the immediate context of $e1'$. Let us assume that e does not happen causally after $c6$; then, we have another uk.p1 condition between $e1'$ and $c6$—a contradiction to the fact that there is no occurrence of a uk.p1 condition between conditions $c4$ and $c6$. Thus, there is no LdCnt(p1) event between $c4$ and $c6$.

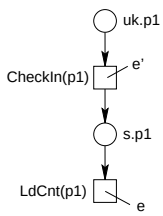


Figure 71: A chain of causes for a LdCnt(p1) event

Step 3, paths from $c4$ to $c6$: Next, we investigate the paths from $c4$ to $c6$ in more detail. In particular, we trace back the way of causes along the path from $c4$ to $c6$ along invariant (2). The automaton from Fig. 72 shows all possible paths:

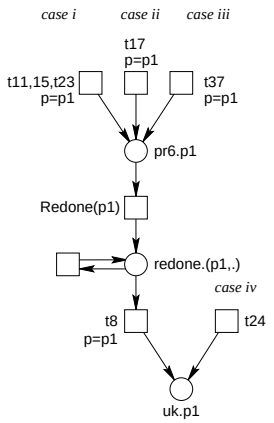


Figure 72: Causes of uk.p1 condition $c6$ along invariant (6)

case i: The process becomes unknown at the server right after a redo procedure of the process, where the redo procedure was terminated without any update because no commit record was encountered during the scan back (transitions t11, t15, and t23 with $p = p1$).

case ii: the process becomes unknown right after the server has updated a continuation from the log to the database during the redo of the process (transition t17 with $p = p1$).

case iii: the process becomes unknown at the server right after the redo, where the redo procedure was terminated by reaching the first log record without doing any update (transition t37 with $p = p1$).

case iv: the process is unknown right after the server recovered from a server crash (transition t24).

Note that all events corresponding to one of the above transitions, happen causally after $c4$ because all places in the automaton are places of invariant (2) and k.p1 does not occur in this automaton.

So, we will consider the path on that invariant from these transitions back to $c4$. For clarity, we deal with each of the above cases separately.

Case i: Let us see what could cause the occurrence of one of the above transitions t11, t15, or t23. Figure 73 shows a backward automaton, which contains only places of invariant (2). Since k.p1 does not occur in this automaton, we know that the chain of causes is initiated by an Ignore(p1) event $e8$, and we know that this event occurs causally after the k.p1 condition $c4$ (note that $e8$ could happen immediately after $c4$). In this automaton, we have indicated some log records that are read by the transitions of the automaton (during the restart procedure). These are represented by dashed lines because these places are not part of the automaton. They are only represented to high-light the log records read on this path.

First of all, we know that, on this path, one of the log records $(l+1, o, p1)$, $(l+1, cnt, p1)$, or $(l+1, dbu, p1)$ was read for some l. Since the Ignore event $e8$ happens causally after $c4$, and therefore also causally after the commit event $e1'$, we know by Property 5 (sequence numbers are increasing) that $m \geq k$. Since the counter i of the automaton is m initially and is only decreased by all events, we know $m \geq l$. Moreover, we know that, on each path of the automaton, a log records with sequence number $i + 1$ is read for each i with $m \leq i < l$; and we know that none of these log records are records of process $p1$ (cf. condition $p' <> p1$). By the uniqueness of sequence numbers of log records (Property 6), by the existence of the log record $(k + 1, cmt, p1)$, and by $m \geq k$, we know $l \geq k$.

Now, let us investigate, where the above log records come from. This is represented in Fig. 74. The

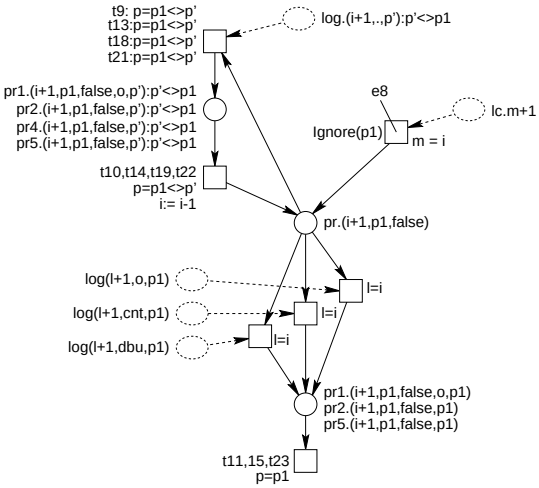


Figure 73: Automaton for *case i*

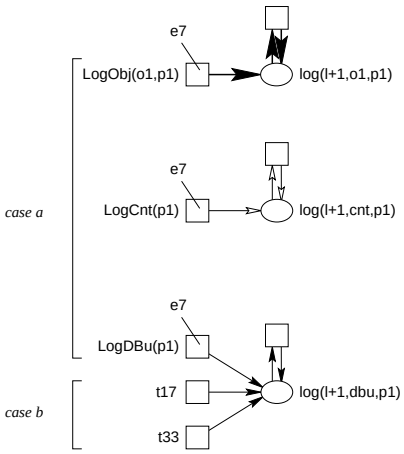


Figure 74: Write operations for log records in *case i*

update record and the continuation record can only originate from a LogObj(o1,p1) resp. LogCnt(p1) event (with possibly some read accesses on these log records in between). The situation for the DBu record is a little bit more involved: This record could either be written by the process itself by a LogDBu(p1) operation; but it could also be written during a redo of the process or during the recovery of the server (transitions t17 and t33).

We first consider the case (*case i.a*) in which the log record is written by a log event $e7$ of process $p1$. Then we will show that the other case (*case i.b*) is impossible in our situation.

Case i.a: Now, we show that, for *case i.a*, we have an event $e6$ that updates the continuation of event $e1$ in the database (see Fig. 68 on 63) and happens causally after $e1'$ and causally before $c6$ (and thus before $c1$)—which finishes the proof of *A.1.2.2.1* in this case.

First of all, we know that the log event $e7$ occurs causally after $e1'$: The sequence number of the written log record is $l + 1$, and we have $l \geq k$. By Property 5 (Increasing sequence numbers), we know that $e1'$ occurs causally before $e7$. Remember that $k + 1$ is the sequence number of the commit record written by $e1'$

Next, we there is a path of program causality from $e1'$ to $e7$ on which a UpdCnt(p1) occurs: By assumption, we know that there is no LdCnt(p1) event between $e1'$ and $c6$. In particular, no LdCnt(p1) occurs between $e1'$ and $e7$ (since $e7$ happens causally between $e1'$ and $c6$ by construction). By invariant (1) $\mathsf{a}[p1]+\mathsf{cr}[p1]+\mathsf{s}[p1] = 1$, we know that there is a path from $e1'$ to $e7$ on which only one of the places of the invariant occur. Moreover, we know, by the above argument, that no LdCnt(p1) event occurs on this path. All possible paths between an commit event and a LogObj(o1,p1), a LogCnt(p1), or a LogDBu(p1) event are represented in the automaton of Fig. 75. The automaton shows that there is an UpdCnt(p1) event $e6$ on this path—note that each occurrence

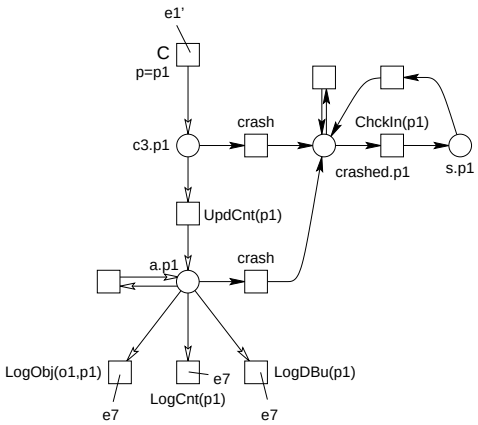


Figure 75: Automaton for commit followed by a log operation

of an UpdCnt(p1) event has an condition cnt.p1 in its immediate context following it. Moreover, there is a path of program causality form $e1$ via $e1'$ to $e6$ and $e6$ happens causally before $e4$. So $e6$ is the event required on the right hand side of requirement *A.1.2.2.1*. Since no LdCnt(p1) event occurs between $e1'$ and $c6$, and $e6$ happens causally between $e1'$ and $c6$, we also know that no LdCnt(p1) event happens causally between $e1'$ and $e6$—as required in *A.1.2.2.1*.

Case i.b: Now, we show that *case i.b* is impossible in the considered situation. Let us consider the occurrence of a t17 or a t33 event with $p = p1$. First of all, t17 produces a token $p1$ on place pr6, or t33 produces some token on sr. By invariant (2), and by the automaton shown in Fig. 73, we know that t17 or t33 must occur before the Ignore(p1) event (pr6.p1 and sr do not occur in this automaton).

Now, let us investigate the path from the occurrence of an t17 or t33 event with $p = p1$ to the Ignore(p1) event – on the above invariant. This path is shown in Fig. 76. This automaton shows that there is a uk.p1 condition $c7$ on each path. This gives us another uk.p1 condition between $c4$ and $c6$ and contradicts our

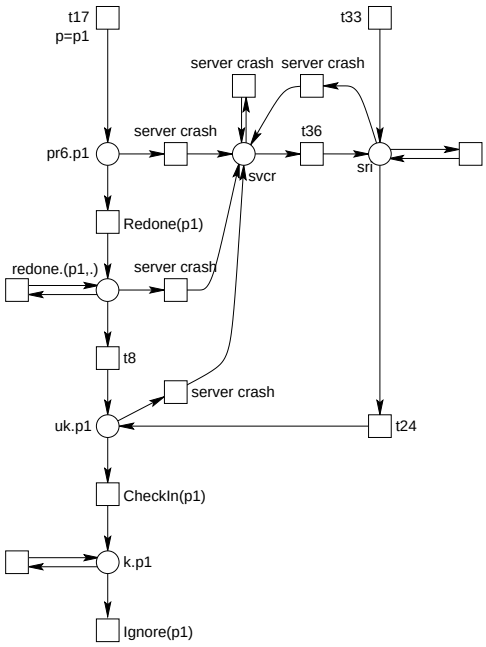


Figure 76: Automaton for a path from t17 or t33 with $p = p1$ to Ignore(p1)

assumption that $c6$ is the first occurrence of an uk.p1 condition on the path from $c4$ to $c5$. Thus, *case i.b* is impossible, which finishes the proof of *case i*.

Case ii: Now, let us assume that the event which caused the uk.p1 condition is t17 with $p = p1$. Fig. 77 shows the chain of causes for this event. Again, we have only places of the above invariant (2) in this automaton. So the Ignore(p1) event happens causally after condition $c4$. Thus, we know $m \geq k \geq j$ by Property 5. Remember that $j+1$ is the sequence number of the continuation record written by event $e5$. On each path of this automaton, there exists an event that reads a log record $(l+1, cnt, p1)$ and this continuation is written to the database by the next event. Let us call this event $e6$. In the following, we will show that this is the update event that is required by *A.1.2.2.1*: It remains to show that there is a path of program causality from $e1$ to the log(l+1,cnt,p1) condition.

Since the automaton only decreases the value of the counter i, we know $m \geq l$. Since on each path, the events that read a log record and the events that decrease i strictly alternate, we know that there is a log record $(i+1, ...)$ for each i with $l \leq i \leq m$. A careful analysis of the transitions t9, t13, t18, and t21 reveals that, on this path, no $(i+1, cnt, p1)$ record is read (the only transition t13 that reads a continuation is restricted to log records of processes p' different from $p1$). By Property 6 (uniqueness of sequence numbers of log records), we know that there is no log record $(i+1, cnt, p1)$ with $l < i \leq m$. On the other hand, we have a log record (see Fig. 68) $(j+1, cnt, p1)$ with $j \leq m$. In combination, we get $j \leq l$.

By the same argument as before (see Fig. 74), we know that log record $(l+1, cnt, p1)$ must have been written to the log file by some LogCnt(p1) event $e7$. Moreover, there is a path of program causality from $e7$ to the condition log.(l+1,cnt,p1). By $j \leq l$, we know that this event happens causally after $e5$. Moreover, we know that there is no LdCnt(p1) between these events. By Property 3, we know that there is a path of program causality from $e1$ to $e7$. Altogether, we have a path of program causality from $e1$ to $e6$. By construction, there are no LdCnt(p1) events between $e1'$ and $e6$, which finishes the proof of *case ii*.

Case iii: Now, we show that *case iii* is impossible. Figure 78 shows the chain of causes for t37 with $p = p1$. By the same arguments as before, we know $m \geq j$ and that no $(i+1, cnt, p1)$ records with $0 < i \leq m$ exits. This is a contradiction to the existence of the log record $(j+1, cnt, p1)$ that is written by the log event $e5$.

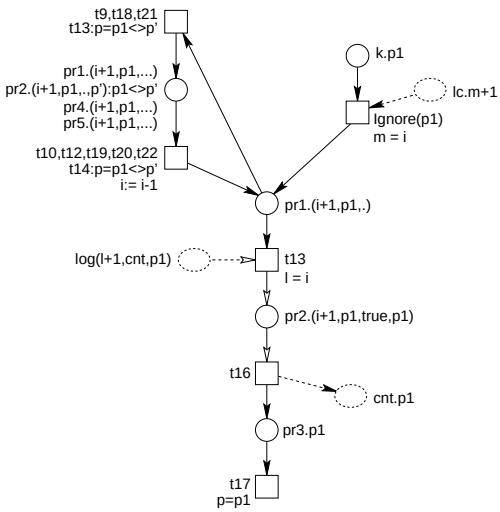


Figure 77: Chain of causes for t17 with $p = p1$

This finishes the proof of *case iii*.

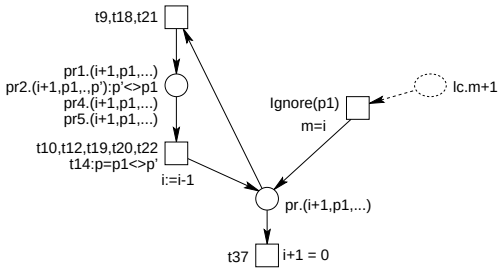


Figure 78: Impossibility of *case iii*

Case iv: Remember that *case iv* was the successful completion of the restart of the server by a t24 event. Let us have a look at the chain of causes of this event. Again, we construct an automaton that has only places of invariant (2) and does not contain a place k.p1. This automaton is shown in Fig. 79. It shows that there exists a path along invariant (2) from the t24 event back to some t36 event, which we call $e7$. By construction (the automaton does not contain a place k.p1), we know that $e7$ occurs causally after the k.p1 condition $c4$. Therefore, we have $m \geq k$ by Property 5. Moreover, we have $i = 0$ when t24 occurs. Due to the strict alternation of reading log records and decreasing the counter i, we know that for each sequence number i with $0 < i \leq m$ a log record $(i, ...)$ is read on the path from t36 to t24. Since sequence numbers are unique (Property 6), we know that the log record $(j+1, cnt, p1)$, which was written by the LogCnt(p1) event $e5$, is read on this path. The only transition of this automaton that can read the log record $(j + 1, cnt, p1)$ are the two instances of transition t29.

We consider both instances of t29 separately: First, we consider the transition at the bottom: In that case, the t29 event reads the log record (j+1,cnt,p1) and the following event t32 updates it to the database. A simple automaton (cf. Fig. 74) shows that there is a program causality from $e5$ via the log record and via t29 to the update event t32.

Second, we consider the transition t29 at the top of this automaton. In that case, the continuation is not updated to the database. Rather, it is ignored by the occurrence of transition t30 or t31. Informally, the log continuation record is ignored because either a continuation of $p1$ has already been updated during this

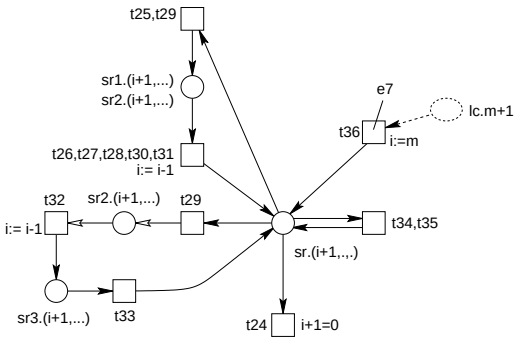


Figure 79: Automaton for t24 event

recovery of the server or there was no corresponding commit event. These situations are shown in Figures 80 (*case vi.a*) and 82 (*case vi.b*).

case iv.a: Let us consider the automaton of Fig. 80 first, which shows the situation when the log record

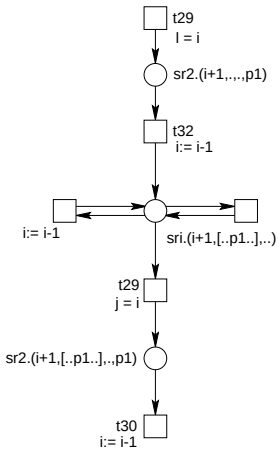


Figure 80: Ignoring an already redone continuation record: *case vi.a*

is ignored due to a prior redo. Event t30 ignores the record because process $p1$ occurs in the set of redone processes (indicated by label $[..p1..]$ at the corresponding places). This automaton shows the chain of causes for transition t30 and t29. We explain the automaton backwards: There is a phase immediately preceding t29 in which $p1$ is in the redone list. In the automaton, however, we do not represent all details; we only represent the events that do not change this situation (one transition stands for the transitions incrementing i; the other for the transitions leaving i unchanged). The only event that can add $p1$ to the redone list is t32 with $p = p1$. So, we have a $(l+1, cnt, p1)$ record with $l > j$. By the automaton of Fig. 74, we know that there is a corresponding LogCnt(p1) event $e8$. By Property 5 and $l > j$ this event happens causally after $e5$. By construction, $e8$ happens causally before $c6$. In particular, there is no LdCnt(p1) event between $e5$ and $e1'$ (by Property 2), and there is no LdCnt(p1) event between $e1'$ and $e8$ by assumption (see Fig. 70). By the same arguments as given in *case i.a*, there exists an intermediate UpdCnt(p1) event $e6$ and a path of program causality from $e1$ to $e6$ (via $e5$): an automaton for a path from $e5$ to $e8$ along invariant (1) as shown in Fig. 81. On each path of this automaton, there exists an UpdCnt(p1) event $e6$ that meets the situation on the right hand side of requirement *A.1.2.2.1*—remember that in the chosen situation, no LdCnt(p1) event occurs between the two LogCnt(p1) events.

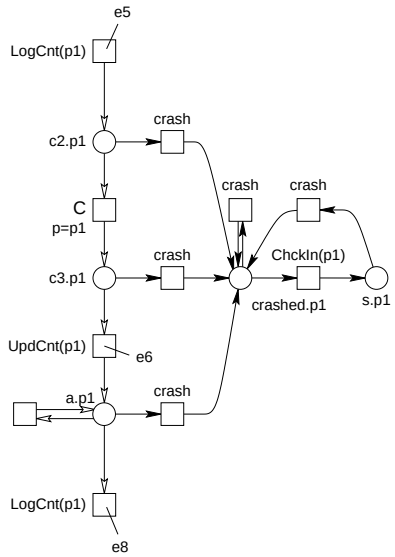


Figure 81: Path between two LogCnt(p1) events with out on which no LdCnt(p1) event occurs

case iv.b: At last, we consider the automaton in Fig. 82, which shows the situation when a log record is ignored because it is not in the commit list. The argument is the similar to the previous one. But, the automaton shows two different possibilities: Either there are no commit records encountered during the restart of the server before the occurrence of t29. In that case, the chain of causes starts with event t36 (*case iv.b.1*). Or it starts with an event t29 that reads a log continuation record and the corresponding transitions t30, t31, or t32 remove $p1$ from the list of committed processes (*case iv.b.2*).

Let us consider *case iv.b.1* first: The automaton says that, in this case, for each i with $j < i \leq m$, there is a log record for $i+1$ and the log record is different from $(i+1, cmt, p1)$ (remember that t36 happens after $c4$ and thus the automaton starts with $i = m \geq k$). On the other hand, we have a log record $(k+1, cmt, p1)$ with $k \geq j$, which violates the uniqueness of log records with the same sequence number(Property 6). Thus, *case iv.b.1* is impossible.

In *case iv.b.2*, we have a log record $(l+1, cnt, p1)$ with $l > j$. In this case, we can proceed as in *case iv.a*: We know that there is LogCnt(p1) event that happens causally after LogCnt(p1) event $e5$; again, by the automaton from Fig. 81, there is an intermediate UpdCnt(p1) event that satisfies the requirement of $e6$ on the right hand side of *A.1.2.2.1.*

7.6 A.1.2.2.2: Correct order of update and loading from database

In this section, we prove requirement *A.1.2.2.2* (see p. 45), which basically says that intermediate updates to the database do not write an older continuation. Figure 83 shows a situation that resembles the situation of the left hand side of requirement *A.1.2.2.2.* We have added the immediate context to the LogCnt event $e5$ and to the commit event $e1'$. By Property 5, we know $j < k$.

First, we deal with the special case $c1 = c$; in that case, we also know $e7 = e6$ because an event immediately preceding a condition is unique in our execution model. Thus, we have a path of program causality from $e1$ via $e6 = e7$ to $c1$. So, we have *case i* of the right hand side of requirement *A.1.2.2.2.*

In the rest of this proof, we assume $c1 \neq c$. Again, by our execution model and the invariant $\mathsf{cnt}[p1] = 1$ (invariant (3), see Table 5 on page 105), we know that $e7$ happens causally after condition c— otherwise there are two concurrent occurrences of cnt.p1 conditions, which violates the invariant.

Now, let us investigate the different events that could occur in the preset of $c1$ and have a program causality to condition $c1$. Figure 84 shows all possible instances of transitions of the Petri net model which update the continuation of process $p1$ in the database:

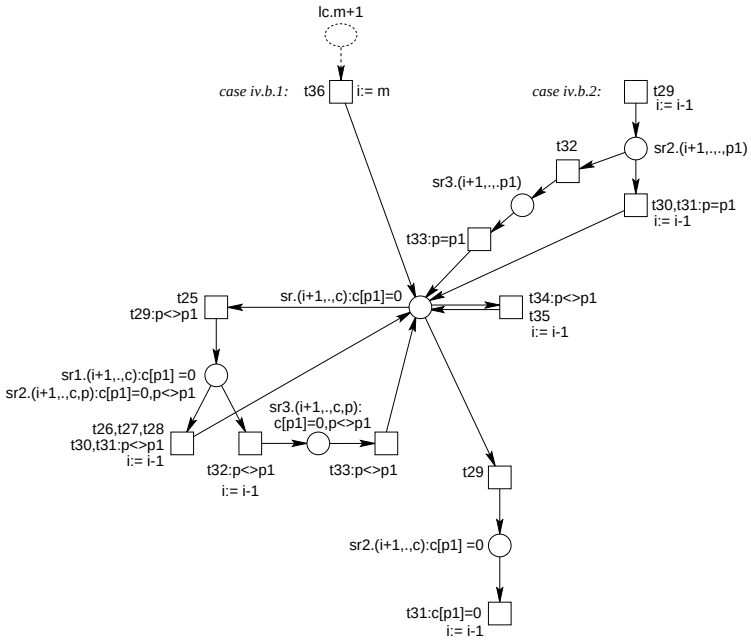


Figure 82: Ignoring an uncommitted continuation record: *case iv.b*

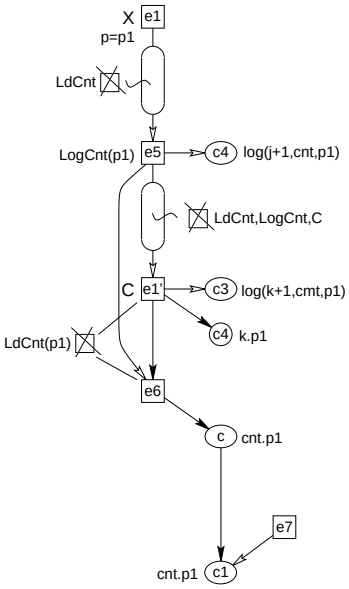


Figure 83: Situation matching the left hand side of *A.1.2.2.2*

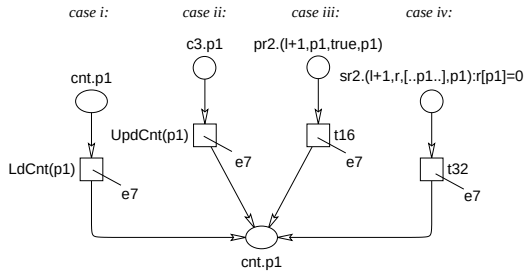


Figure 84: Four different update events for cnt.p1

case i: Event $e7$ is a load continuation event. Note that from the pragmatic point of view, this is not an update event at all; but technically, it is an event which puts a token $p1$ to place cnt and thus must be considered in the proof.

In this case, however, we have a cnt.p1 condition $c2$ in the preset of $e7$ (the continuation loaded) and there is a program causality from $c2$ to $e7$. Moreover, we know that $c2$ occurs causally after c because of the above invariant (3) and because $e7$ happens causally after c,. So, we have the situation of *case ii* of the right hand side of requirement *A.1.2.2.2* (where $e7$ and $e8$ coincide).

case ii: Event $e7$ is a UpdCnt(p1) event. Remember that $e7$ happens causally after c. In particular, $e7$ happens causally after the LogCnt(p1) event $e5$, which will be exploited in the proof for this case below.

case iii: Event $e7$ is a t16 event, which updates the continuation in the database during a redo of process $p1$.

case iv: Event $e7$ is a t32 event, which updates the continuation in the database during the restart of the server.

Since *case i* was already proven above, we need to consider only *cases ii–iv* in the following. We proceed in three steps: First, let u assume (a proof will be given below) that, in *case iii* and *case iv*, we have a LogCnt(p1) $e7'$ event with a program causality to $e7$ that causally happens after the LogCnt(p1) event $e5$. In each of the remaining *cases ii–iv*, we either have a UpdCnt(p1) event $e7' = e7$ which happens causally after $e5$ (in *case ii*) or we have a LogCnt(p1) event $e7'$ that happens causally after $e5$ (in *case iii* and *case iv*).

Second, we know that there exists a LdCnt(p1) event $e8$ with a path of program causality to $e7'$ by applying Property 4. Moreover, on this path only conditions a.p1 occur.

The resulting situation is shown in Fig. 85. On the path of program causality from $e1$ to $e1'$, only conditions a.p1 occur by Property 2.

Third, we prove for this situation, that there is a path of program causality from $e1$ to $c1$ (*case i* of requirement *A.1.2.2.2*), or that condition $c2$ occurs causally after condition c. The arguments are as follows: By invariant (1), we know that all events on the two paths of program causality from $e1$ to $e1'$ and from $e8$ to $e7'$ are sequentially ordered. This gives us the following cases:

- Event $e8$ occurs causally after $e1'$: Since we know that no LdCnt(p1) event occurs between $e1$ and $e6$, we know that $e8$ happens causally after $e6$. Again by invariant $\mathsf{cnt}[p1] = 1$ (invariant (3)), we know that condition $c2$ occurs causally after condition c. This is the situation of *case ii* of requirement *A.1.2.2.2*.

- Event $e8$ occurs on the path from $e1$ to $e1'$; this, however, is not possible because we assumed that there are no LdCnt events in this path.

- Event $e1$ occurs on the path of program causality from $e8$ to $e7'$. This gives us a path of program causality from $e1$ via $e7'$ to $c1$. This is the situation of *case i* of requirement *A.1.2.2.2*.

It remains to show that, in *case iii* and *case iv*, we have a LogCnt(p1) event $e7'$ that occurs causally after $e1'$ and that has a path of program causality to $e7$.

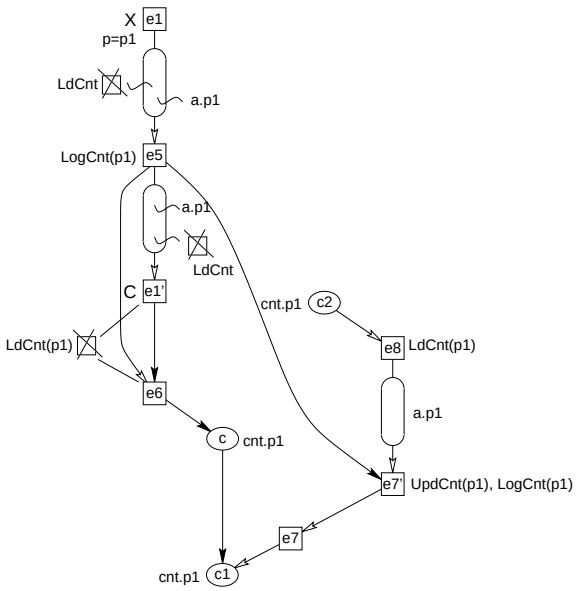


Figure 85: Situation in *cases ii–iv*

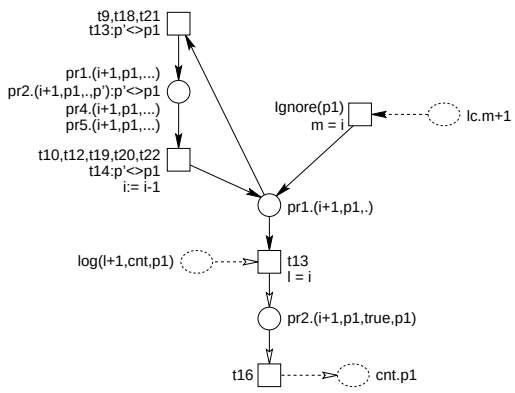


Figure 86: Chain of causes for `t16` (*case iii*)

LogCnt(p1) event $e7'$ in *case iii* We start with the proof of *case iii*. Figure 86 shows the automaton for the chain of causes for the t16 event (update during redo of process $p1$). Note, that this automaton contains only places from invariant (2) and k.p1 does not occur in this automaton. Therefore, event Ignore(p1) occurs causally after the commit event $e1'$ (which has a condition k.p1 in its postset). By Property 5, we know $k \leq m$. Due to the structure of the automaton, we know that on the path from the Ignore(p1) event to t16, for each i with $l < i \leq m$, a log record with sequence number $i+1$ is read. Moreover, for all i with $l < i \leq m$, the read log records are different from $(i+1, cnt, p1)$ (the only transition, t13, reading a $(i+1, cnt, p')$ record is restricted to $p' \neq p1$). By the uniqueness of sequence numbers for log records (Property 6 and the log record log.(j+1,cnt,p1), we have $j \leq l$.

We know (see automaton Fig. 74) that there exists a LogCnt(p1) event $e7'$ that has written the log record log.(l+1,cnt,p1) and there is a path of program causality from this log event to $c1$ (via the occurrence of t13 and t16). By Property 5 and $j \leq l$, we know that $e7'$ occurs causally after the LogCnt(p1) event $e5$. Note that $e5$ and $e7'$ could be identical. This gives us the event $e7'$ as shown in Fig. 85 and finishes the proof of *case iii*.

LogCnt(p1) event $e7'$ in *case iv* Figure 87 shows the automaton for the chain of causes for the t32 event (update during restart of the server). Again, we know that t36 happens causally after the commit event $e1'$ because the automaton contains only places from invariant (3) and k.p1 does not occur in this automaton; thus, by Property 5, we have $k \leq m$.

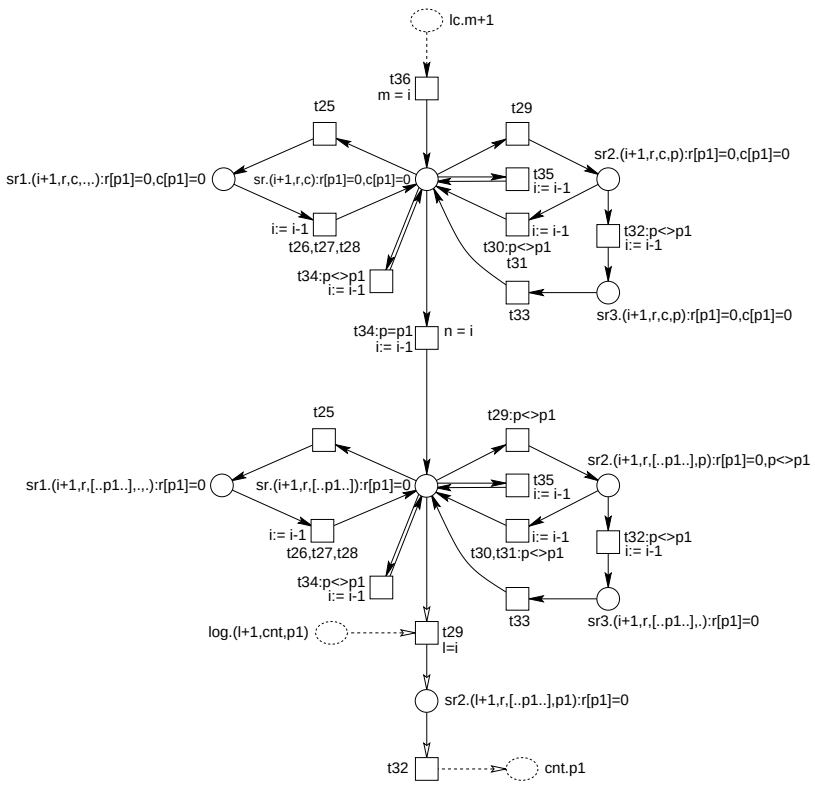


Figure 87: Chain of causes for t32 *case iv*

The automaton basically shows the part of the restart procedure in which the continuation of $p1$ is not yet listed as redone (otherwise t32 would not have occurred). The lower part of the automaton shows the part from the commit event corresponding to the updated log record; in this part, no continuation record for $p1$ is read (except for $(l+1, cnt, p1)$). The upper part of the automaton shows the behavior before the

commit event for process $p1$ has been encountered. Note that, in this part, continuation records for process $p1$ could be read; but there are no commit records encountered for $p1$.

First of all, we know that there exists a LogCnt(p1) event $e7'$ that writes $(l+1, cnt, p1)$ by the automaton in Fig. 74, and we know that there is a path of program causality from $e7'$ to the t32 event $e7$. It remains to show that $e7'$ occurs causally after $e5$. By Property 5, it is sufficient to show $j \leq l$. To this end, we consider the automaton from Fig. 87 again. The only event in this automaton that reads a commit record for process $p1$ is t34 with $p = p1$ (in the middle of the automaton); let $n+1$ be the sequence number of this log record. By $m \geq k$, and the uniqueness of sequence numbers of log records (Property 6), we have $n = k$ or $k \leq l$. In case $k \leq l$, we immediately have $j \leq l$ because $j \leq k$ by assumption. It remains to show $j \leq l$ in case $n = k$ (and $k > l$): from the automaton, we know that all log records with sequence number $i+1$ for $l < i \leq k$ are different from $(i+1, cnt, p1)$. By Property 6, by the existence of the log record $(j+1, cnt, , p1)$, and by $j \leq k$, we know $j \leq l$.

7.7 B.3.1: Two cases of data causality

In this section, we verify requirement *B.3.1* (see p. 49): A path of data causality from a write event $e1$ on some object $o1$ to some access event $e2$ on the same object $o1$ is either via a local copy cp.(p1,o1,wr) or there is an intermediate update to the database and a LdCpy(o1) event.

Analog to the proof of requirement *A.2.1*, this property can be proven by an automaton for data causalities between the two events. The automaton for paths of data causality shown in Fig. 88 verifies requirement *B.3.1*. The occurrence of $e2$ on the left hand side of the automaton corresponds to *case ii* on the right

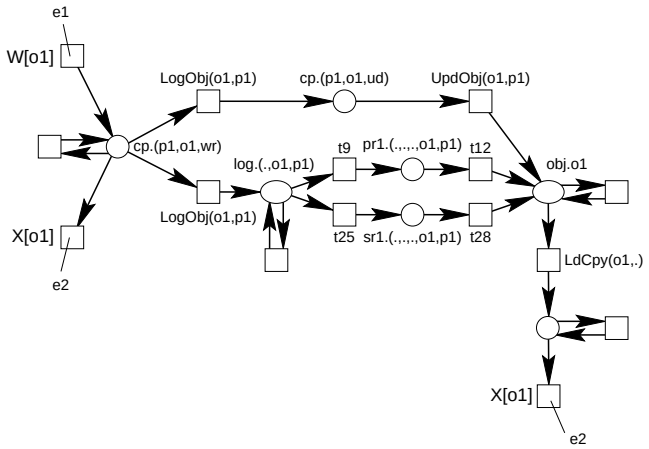


Figure 88: Automaton for paths of data causality from W[o1] events to some access event X[o1]

hand side of requirement *B.3.1*; the occurrence of $e2$ on the right hand side of the automaton corresponds to *case i* on the right hand side of requirement *B.3.1*.

7.8 B.3.2: Updated object implies direct commit

In this section, we verify requirement *B.3.2* (see p. 49): If there is a path of data causality from a write event to an event that updates the database, then the write event is a directly committed event. The proof is analog to the proof of Requirement *A.2.2* (see p. 57).

Again, there are three different ways in which an object can be written to the database: by an update event UpdObj during the execution of the commit operation, by an update event t12 during the redo procedure of a single process, and by an update event t28 during the restart of the server. Formally, these three cases are captured by the automaton of Fig. 89. This automaton captures all possible paths of data causality from a W[o1] event $e1$ to an obj.o1 condition $c1$. From this automaton, we get the following cases, which are graphically represented in Fig. 90:

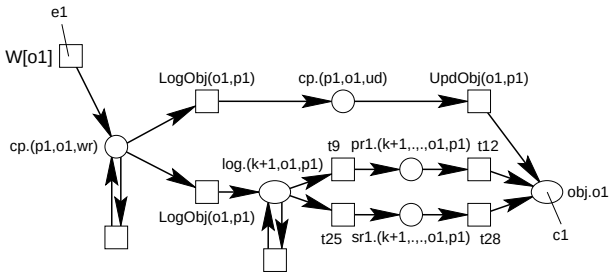


Figure 89: Automaton for paths of data causality from W[o1] events to obj.o1 conditions

***case i*:** The path is via a LogObj(o1,p1) event $e2$ and the update event $e3$ is a UpdObj(o1,p1) event.

***case ii*:** The path is via a LogObj(o1,p1) event $e2$, log records $(k+1, o1, p1)$, and an t12 event $e3$.

***case iii*:** The path is via a LogObj(o1,p1) event $e2$, log records $(k+1, o1, p1)$, and an t28 event $e3$.

In the following, we prove that there exists a direct commit event $e1'$ for $e1$ in each of the above cases.

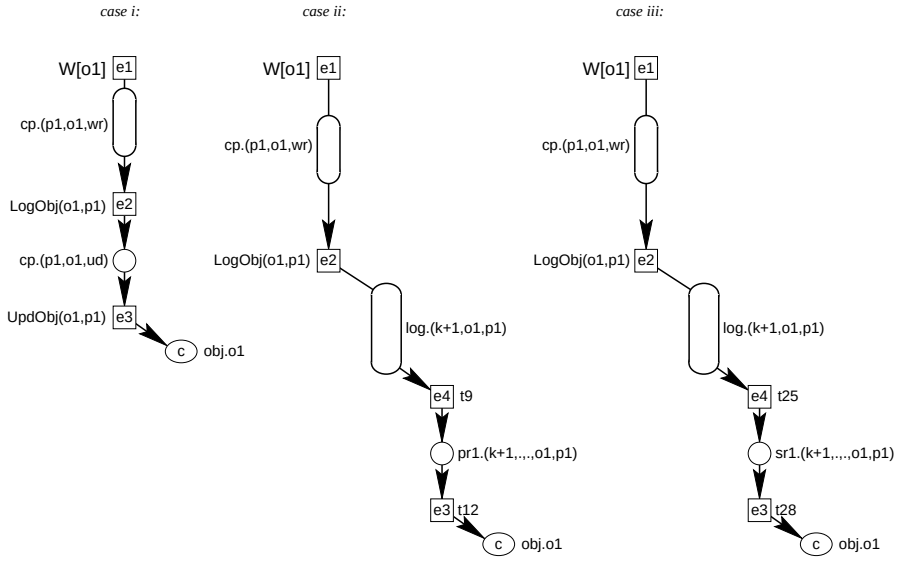


Figure 90: Req. *B.3.2*: Three cases

***Case i*:** First of all, we know, by Property 9.1, that is nod LdCnt(p1) event between 1 and $e3$. By Property 3, there is a path of program causality on which no LdCnt event occurs from $e1$ via $e2$ to $e3$.

Next, we show that, on the path of program causality from $e2$ to $e3$, a commit event occurs. The automaton from Fig. 91 shows all possible paths of program causality from a LogObj(o1,p1) event to a UpdObj(o1,p1) event. On all these paths, a commit event occurs—without a LdCnt event occurring before. This commit event $e1'$ is a direct commit event for event $e1$.

***Case ii and iii*:** By the same arguments as above (Property 9.1 and 3), we know that there is a path of program causality from event $e1$ to $e2$ in *case ii* and *case iii*. Below, we will show, for both cases separately, that there exists a commit event $e5$ of process $p1$ that happens causally after event $e2$ with no LogCnt(p1) event in between as shown in Fig. 92. Then, we know by Property 1 that there is a path of program causality

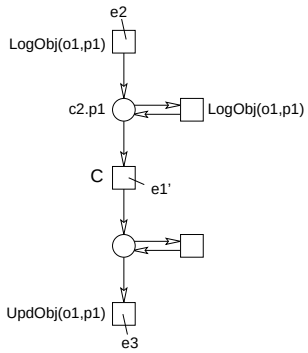


Figure 91: Automaton for paths of program causality from a LogObj(o1,p1) event to a UpdObj(o1,p1) event.

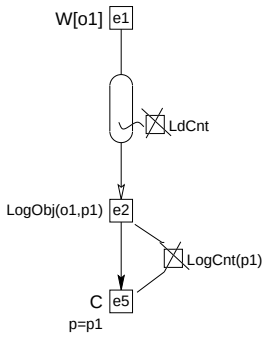


Figure 92: Proof obligation for *case ii* and *case iii*.

from $e2$ to $e5$ on which no LdCnt event occurs. Thus, event $e5$ from Fig. 92 is a direct commit event for $e1$.

It remains to show that the commit event $e5$ from Fig. 92 exists in *case ii* and *case iii* and that there is no LogCnt(p1) event between $e2$ and $e5$.

***Case ii*:** We start with verifying the existence of event $e5$ from Fig. 92 for *case ii*. To this end, we consider the automaton from Fig. 93. This automaton shows that there is a commit record $(l+1, cmt, p1)$ for process $p1$ with a sequence number $l+1$ greater than $k+1$, which is written by a commit event $e5$. By Property 5, this commit event $e5$ happens causally after $e2$. Moreover, the automaton shows that, for each i with $k < i < l$ there is a log record with sequence number $i+1$ that is different from $(i+1, cnt, p1)$. By the uniqueness of the sequence numbers of log records (Property 6), we know that there is no LogCnt(p1) event that writes a log record $(i+1, cnt, p1)$ with $k < i < l$. By Property 5, we know that there is no LogCnt(p1) event between $e2$ and $e5$.

***Case iii*:** At last, we verify the existence of event $e5$ from Fig. 92 for *case iii*. The automaton from Fig. 94 shows the chain of causes for the t28 event $e3$. The automaton shows the existence of a commit record $(l+1, cmt, p1)$ for process $p1$ and the existence of the corresponding commit event $e5$. Moreover, it shows that all intermediate log records for sequence numbers with $i+1$ with $k < i < l$ are different from $(i+1, cnt, p1)$. By the same arguments as in *case ii* we know that the commit event $e5$ happens causally after $e2$ and that there is no LdCnt(p1) event between $e2$ and $e5$.

7.9 B.3.3: Access of same local copy implies same execution phase

Requirement *B.3.3* (see p. 49) says: If there is a path of data causality from a write event $e1$ to an access event $e2$ or a LogObj event $e2$ on which only cp(p1,o1,.) conditions occur, then there is also a path of program

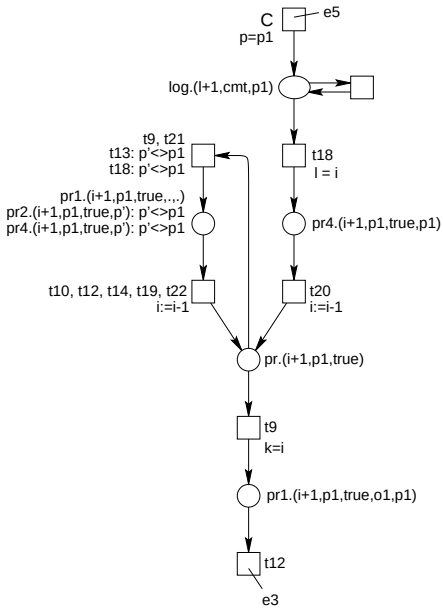


Figure 93: Automaton for chain of causes of t12 event.

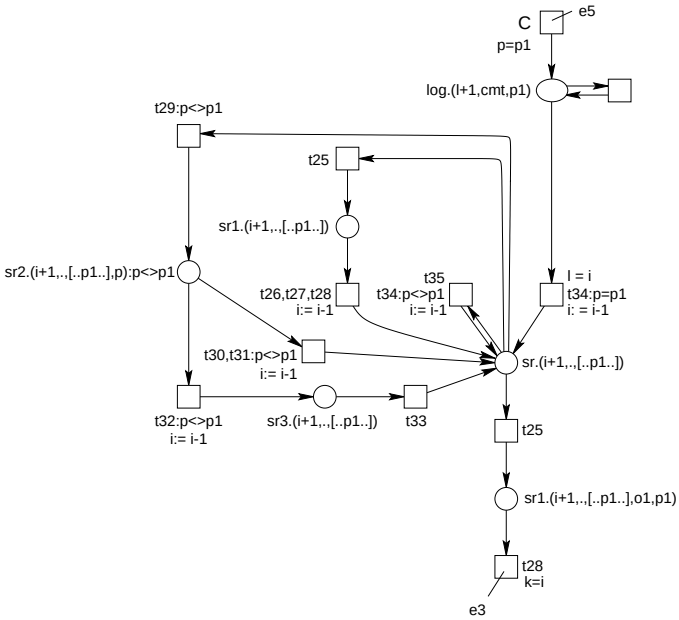


Figure 94: Automaton for chain of causes of t28 event.

causality from $e1$ to $e2$ and there is no LdCnt event on this path.

It will turn out that we need a slightly more general version of this property for verifying other requirements. Therefore, we prove the more general Property 9 in Sect. 8.3. Requirement *B.3.3* is an immediate consequence of Property 9 and Property 3.

7.10 B.1.1: Context of a write event

In this section, we proof requirement *B.1.1*, which talks about the context of a directly committed write event (see p. 50).

Let us consider a situation that meets the right hand side of requirement *B.1.1*. From the Petri net model, we know that there exists a cp.(p1,o1,wr) condition in the postset of event $e1$ and a condition wlckcnt.(p1,0) in the postset of the commit event $e1'$. By invariant (7) (see Table 5 on page 105), we know that both conditions cannot be concurrent. Thus, there exists some event $e6$ that removes the token $(p1, o1, wr)$ from place cp as shown in Fig. 95. The only transitions that remove this token (and do not add it again) in the Petri net model are the LogObj(o1,p1) transition and the crash transition of process $p1$. An occurrence of a crash transition for process $p1$ is impossible for the following reason: Let us assume that $e6$ is a crash event of process $p1$. Then, $e6$ has a condition cr.p1 in its postset; by Property 2, there exists also a chain of a.p1 conditions between $e1$ and $e1'$. Since $e6$ occurs causally between $e1$ and $e1'$, this implies that a condition a.p1 and a condition cr.p1 occur concurrently; this, however, is a contradiction to invariant (1). Thus, $e6$ cannot be a crash event of process $p1$. Altogether, we know that $e6$ is a LogObj(o1,p1) event. Since this event has an a.p1 condition in its immediate context, we know by invariant (1) that $e6$ occurs on the path of program causality between $e1$ and $e1'$. In particular, $e1'$ is a direct commit event for $e6$. Thus, we can

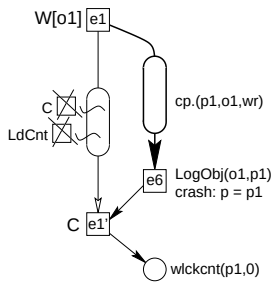


Figure 95: *B.1.1*:Situation constructed from the assumption.

apply Property 10, which gives us the situation shown in Fig. 96. Note that we have renamed some events in order to avoid name clashes. It remains to show that $e1$ occurs on the path of program causality from $e4$ to $e5$: By invariant (5), we know that either $e4$ occurs on the paths of cp(p1,o1,wr) conditions from $e1$ to $e6$ or $e1$ occurs on the path of cp(p1,o1,wr) conditions from $e4$ to $e6$. The first case, however, is impossible because $e4$ has a w2.(p1,o1) condition in its preset. Thus, an occurrence of $e4$ on the cp(p1,o1,wr) path violates invariant (5).

It remains to consider the case that $e1$ occurs on the cp(p1,o1,wr) path from $e4$ to $e6$. Since $e1$ has an a.p1 condition in its context, $e1$ occurs on the a.p1 path from $e4$ via $e5$ to $e6$ by invariant (1). Since there are only LogObj(o1,p1) events on the path from $e5$ to $e6$, by assumption, we know that $e1$ occurs before $e5$. Altogether, we have the situation on the right hand side of *B.1.1*.

7.11 B.1.2: Context of an access event

In this section, we verify requirement *B.1.2*, which talks about the context of a directly committed access event $e1$ (see p. 51). This context is similar to the context of a write event—but a lock need not be kept until the commit event.

We distinguish two cases:

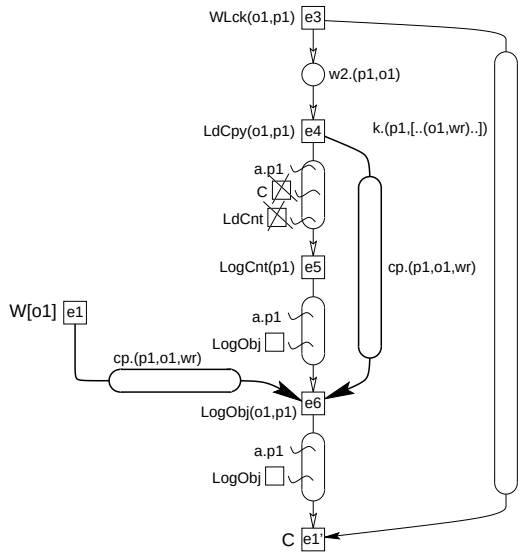


Figure 96: *B.1.1*: Situation after application of Property 10.

1. The access event $e1$ is a write event. In that case, requirement *B.1.1* applies, and we get the situation on the right hand side of *B.1.1*, which also meets the situation on the right hand side of *B.1.2* (with $e5 = e1'$).

2. The access event $e1$ is a read event. We will prove requirement *B.1.2* for this case below.

By Property 7, we know that there exists a `LdCpy(o1,p1)` event $e4$ and a `RLck(o1,p1)` event $e3$ as shown in Fig. 97. By Property 9.1, there are no `LdCnt(p1)`, no `RelLck(p1)`, and no `UnLck(o1,p1)` events between $e3$ and $e1$. By Property 3, there is a path of program causality from $e4$ to $e1$ as shown in Fig. 97. By Property 9.3, we get the right hand side of requirement *B.1.2*: an event $e5$ between $e1$ and $e1'$ with a path of `k.(p1,[..(o1,.)..])` conditions from $e3$ to $e5$.

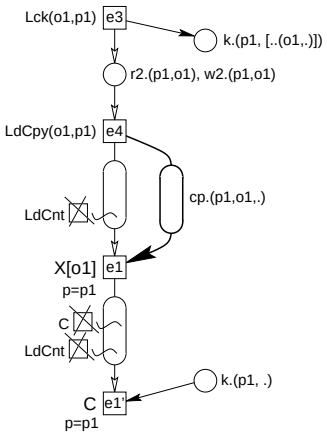


Figure 97: *B.1.2*:Situation constructed from the assumption.

7.12 B.1.3: Write locks are exclusive

In this section, we verify requirement *B.1.3* (see p. 51): Write locks held by the server are exclusive.

Let us consider a situation that meets the left hand side of *B.1.3*, and let us assume $e3 \not\leq e2$, $e4 \not\leq e1$, and $e1 \neq e2$ (i.e. no case of the right hand side of *B.1.3* applies). Then, we have one of the following two cases:

1. Event $e2$ occurs concurrently to some k.(p1,[..(o1,wr)..]) condition c.

   Since $e2$ is some Lck(o1,p2) event, $e2$ has a lock.(o1,m) condition c' in its preset for some m. This condition c' is also concurrent to c. In this situation, invariant (9)

   $$\mathsf{locks}(o1, m) \Rightarrow \forall \mathsf{p}, \forall m' : \mathsf{k.(p}, m') \Rightarrow m'[(o1, wr)] = 0$$

   is violated. Thus, this case is impossible.

2. Event $e1$ occurs concurrently to some k.(p2,[..(o1,.)..]) condition c.

   Since $e1$ is some WLck(o1,p2) event, $e1$ has a lock.(o1,[]) condition c' in its preset. Again, this condition c' is concurrent to c. Thus, invariant (8)

   $$\mathsf{locks}(o1, []) \Rightarrow \forall \mathsf{p}, \forall m : \mathsf{k.(p}, m) \Rightarrow m[(o1, x)] = 0)$$

   is violated. Thus, this case is also impossible.

Altogether, we know that either $e3 \leq e2$ (Req. *B.1.3 case i*), $e4 \leq e1$ (Req. *B.1.3 case ii*), or $e1 = e3$ (Req. *B.1.3 case iii*) holds true.

7.13 B.2.1: Two cases of conflicting events

In this section, we verify requirement *B.2.1* (see p. 52).

Let consider a situation that meets the left hand side of requirement *B.2.1*. Obviously, events $e1$ and $e2$ meet the situation on the left hand side of requirement *B.1.1* and events $e1'$ and $e2'$ meet the situation on the left hand side of requirement *B.1.2*. Applying these requirements gives us the situation shown in Fig. 98.

By Requirement *B.1.3*, we have one of the following three cases:

$e3 = e3'$ By Property 8, we know also $e4 = e4'$. Thus, we have $e4' = e4 \leq e1 \leq e1'$. By invariant (5), we know that $e1$ occurs on the path of data causality from $e4'$ to $e1'$ on which only cp(p1,o1,.) conditions occur. In particular, we have a path of data causality from $e1$ to $e1'$. Thus, we have the situation of *case i* on the right hand side of requirement *B.2.1*.

$e2 \leq e3'$ This situation immediately meets *case ii* on the right hand side of requirement *B.2.1*.

$e2' \leq e3$ This situation is impossible because there is a cyclic causality $e2' \leq e3 \leq e1 \leq e1' \leq e2'$.

7.14 B.2.2.1: Update database before load

In this section, we proof requirement *B.2.2.1* (see p. 53). Figure 99 shows the assumption of requirement *B.2.2.1*—the left hand side of the requirement. Basically, we have to show that there is an event $e7$ that updates the object of the LogObj(o1,p1) event to the database causally between the commit event $e1'$ and the Lck(o1) event $e2$. Moreover, there exists an event $e8$ that writes a DBu record for process $p1$ causally after $e7$ and there are no Lck(o1) events between $e1'$ and $e8$ (see page 53).

The proof of *B.2.2.1* is quite involved because different update mechanisms may interfere: an update by the process itself, an update during the redo procedure of a process, and an update during the restart of the server. These different update mechanisms will be reflected in the cases *i–iii* below.

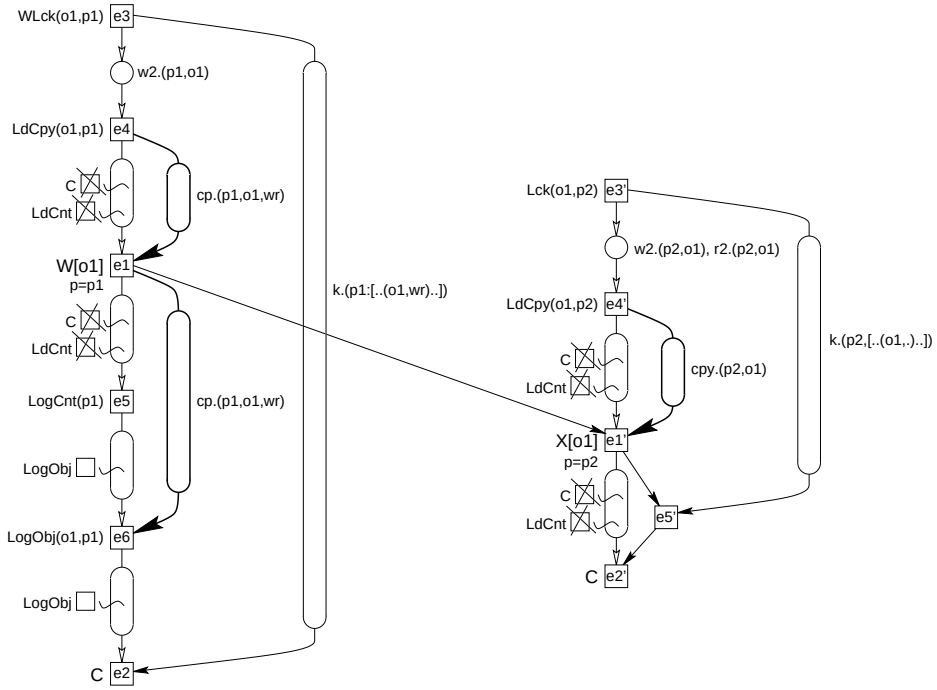


Figure 98: *B.2.1*: Situation after application of *B.1.1* and *B.1.2*.

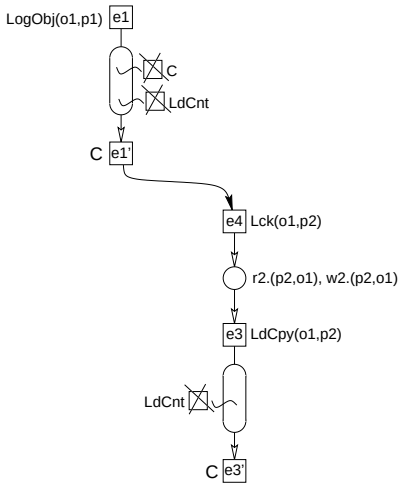


Figure 99: Assumption of Requirement *B.2.2.1*

Step 1, adding context for process $p1$ and process $p2$: We start by adding some context to the situation shown in Fig. 99. Note that event $e1$ and $e1'$ meet the situation on the left hand side of Property 10. Thus, we know that we also have the situation on the right hand side of Property 10. This situation is shown in Fig. 100, where we have added some context to the lock event $e4$, to the commit event $e1'$, and to the log event $e6$ and $e1$, which follows immediately from the context of the corresponding transition in the Petri net.

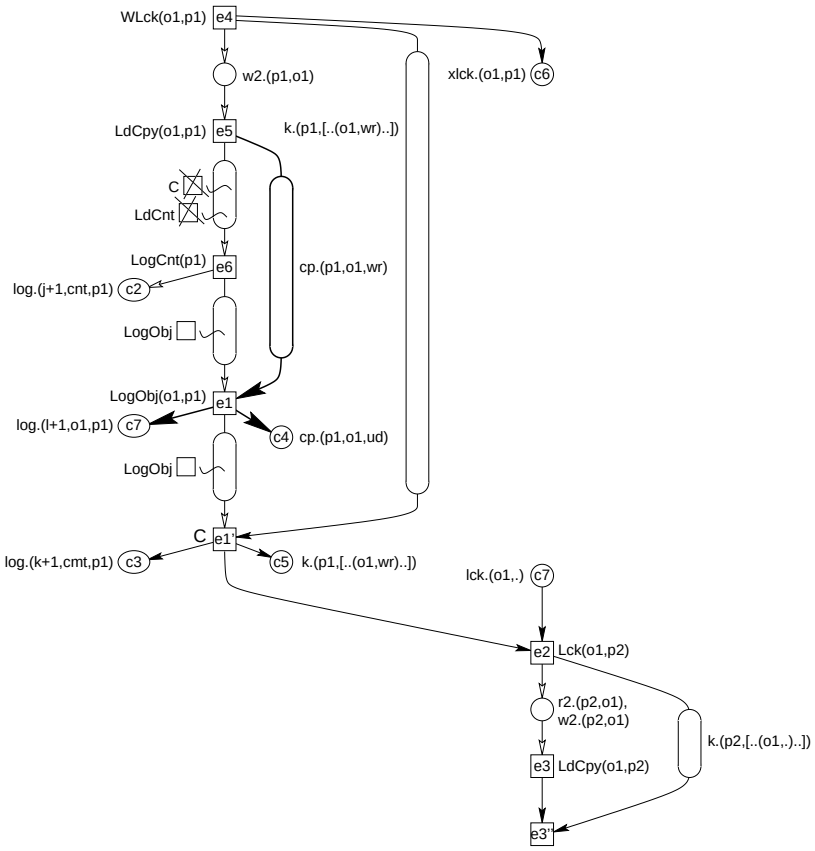


Figure 100: Assumption of *B.2.2.1*: Added context

By Property 2, there is no LdCnt(p2) event between $e3$ and $e3'$. By Property 9.1, there is no RelLck(p2) and no UnLck(o1,p2) event between $e4$ and $e3$. Thus, we can apply Property 9.3, which gives us an even $e3''$ that occurs causally after $e3$ and there is a chain of k.(p2,[..(o1,.)..]) from $e2$ to $e3''$ as shown in Fig. 100. Moreover, we have added the lck.(o1,.) condition in the preset of $e2$, which immediately follows from the context of a lock transition in the Petri net model.

By Property 5 (increasing sequence numbers of log records), we also know $j \leq l \leq k$.

Step 2, the path of causality between $c6$ and $c7$ By invariant (4), $pr_1(\mathsf{xlck} + \mathsf{lck})[\mathsf{o1}] + \mathsf{svcr} + \mathsf{sri} = 1$, we know that there is a unique chain of conditions of this invariant from condition $c6$ to condition $c7$. In particular, there exists a first occurrence of a lck.(o1,.) condition $c8$ on this chain. Informally, this is the point at which the lock of process $p1$ on object $o1$ that was acquired by $e4$ is released at the server. In the following, we will show that the object $o1$ corresponding to the LogObj(o1,p1) event $e1$ is updated to the database before this point.

Step 3, the different cases Next, let us investigate the possible paths on the above chain from $c6$ to $c8$ (the first lck(o1,.) condition). Figure 101 shows an automaton with all possible paths along invariant (4) from the xlck.(o1,p1) condition $c6$ to the lck.(o1,.) condition $c8$. The occurrence of the first condition

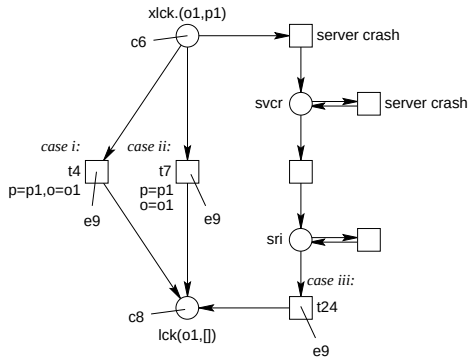


Figure 101: Three cases

lck.(o1,.) on this path is caused by an occurrence of either transition t4, transition t7, or transition t24; we denote the corresponding event by $e9$. Let us discuss the different cases in more detail:

***case i*:** The lock at the server is released by an occurrence of transition t4 (see Fig. 7 on page 12). This is a release of the lock by the server after a corresponding request by process $p1$.

***case ii*:** The lock at the server is released by an occurrence of transition t7 (see Fig. 10 on page 14). This is a release of the lock after the process was redone by the server and is now going to be ignored.

***case iii*:** A crash of the server releases all locks. A new lock operation, however, is only possible after a successful restart of the server (the occurrence of transition t24, see Fig. 16 on page 17 and following figures).

Note that, in particular, there are no Lck(o1,.) events between $e4$ and $e9$ because each lock event Lck(o1,.) has an occurrence of condition lck(o1,.) in its preset. Such conditions, however, do not exist between $e4$ and $e9$ by construction of $e9$ and invariant (4).

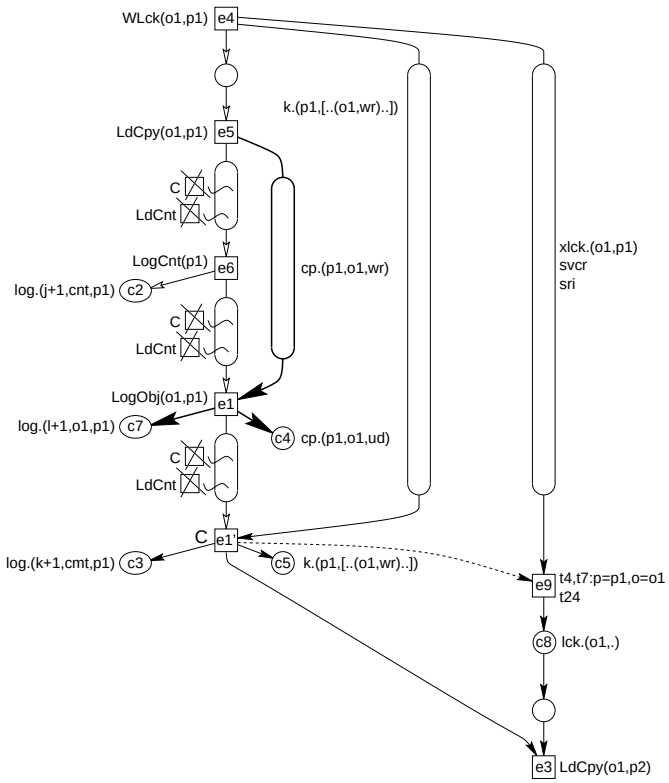
Step4, $e9$ occurs after $e1'$ Before dealing with each of the above cases individually, we summarize what we have so far in Figure 102. Where the indicated dashed causality from $e1'$ to $e9$ needs a further argument, which will be given below. Moreover, we know that no Lck(o1,.) event can occur between $e4$ and $e9$ (see argument above).

Next, we give arguments for the dashed causality from $e1'$ to $e9$: In each of the above cases, event $e9$ has one of the conditions shown in Fig. 103 in its context. By invariant (2),

$$\mathsf{k}[p1] + \mathsf{uk}[p1] + \mathsf{rel}[p1] + \mathsf{pri}[p1] + \mathsf{red}[p1] + \mid \mathsf{sri} + \mathsf{svcr} \mid = 1$$

event $e9$ cannot occur between $e4$ and $e1'$—otherwise are two different concurrent occurrences of condition corresponding to invariant (2): one condition labeled k.(p1,[..(o1,wr)..]) and the other condition labeled by one of the conditions shown in Fig. 103. Since, by construction, $e9$ occurs causally after $e4$, event $e9$ occurs causally after the commit event $e1'$. And there is a chain of conditions of invariant (2) from $c5$ to $c9$.

Step 5, no intermediate condition uk.p1 At last, we show that there is no intermediate uk.p1 condition between $c5$ and $c9$. To this end, we give an automaton for the paths from a k.(p1,[..(o1,wr)..]) condition to a uk.p1 condition. This automaton is shown in Fig. 104. On each path, there is either a t24 event, or there is a t4 or the a t7 event with $p = p1$ and $o = o1$. Let us call this event $e10$. Each of these events has a sri.(0,.,.) condition or a xlck.(o1,p1) condition in its preset, and has neither a xlck.(o1,p1) nor a sri condition



$j < l < k$

Figure 102: Situation after three steps

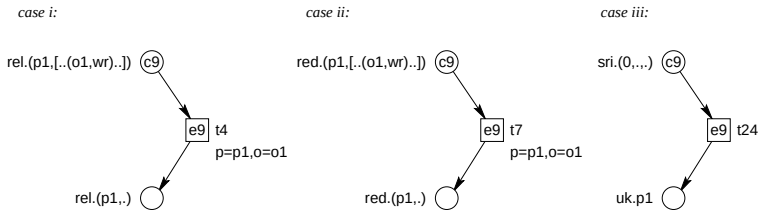


Figure 103: Context of event $e9$ in each case

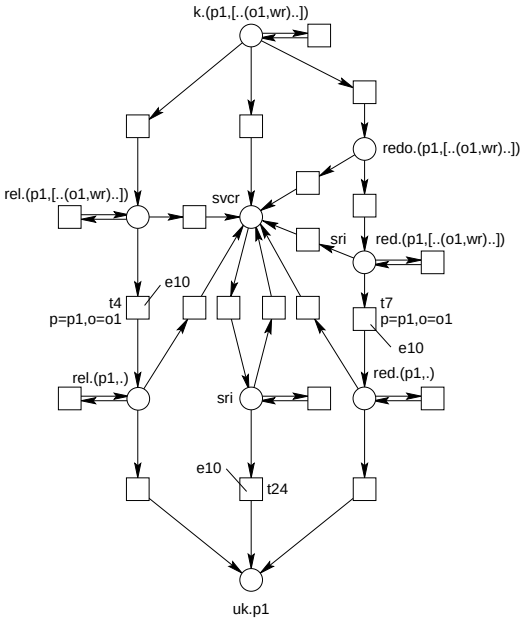


Figure 104: Automaton for reaching uk.p1 from k.(p1,[..(o1,p1)..])

in its postset. Thus, event $e9 = e10$ or $e10$ happens causally after $e9$. In both cases, the uk.p1 condition occurs causally after $e9$. Thus, there is no uk.p1 condition between $e1'$ and $e9$.

Now, the situation is similar to the situation in the proof of requirement *A.1.2.2.1*. This allows us to re-use some of the arguments. For example, we know that there is no LdCnt(p1) event between $e1'$ and $e9$. The arguments are exactly the same as in *A.1.2.2.1*: basically, an intermediate LdCnt(p1) event would imply a uk.p1 condition between $e1'$ and $e9$ (see Step 2 of the proof of *A.1.2.2.1* and the corresponding Automaton on page 64).

In the rest of the proof of *B.2.2.1*, we consider each of the above cases separately.

Case i: $e9$ is a t4 event with $p = p1$ As already mentioned before, event $e9$ was caused by a RelLck(p1) event in this case. The chain of causes for this event is shown in Fig. 105. Let us call the RelLck(p1) event

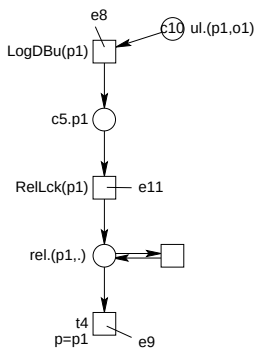


Figure 105: Release operation causes t4; release caused by LogDBu(p1) event

$e11$. Since $t4$ occurs causally after $e1'$ and due to the chain of conditions rel.(p1,.) between $e11$ and $e9$, we know that $e11$ also occurs causally after $e1'$ by invariant (2) (see above). The RelLck(p1) event is caused by a LogDBu(p1) event $e8$. By invariant (1), and the intermediate condition c5.p1, we know that also this

event occurs causally after $e1'$. This LogDBu(p1) event, in turn, has a ul.(p1,o1) condition $c10$ in its preset (a LogDBu(p1) event is deferred until all copies are set to unlocked on the process's side).

Now, let consider our situation from Fig. 102 again. We have a cp.(p1,o1,ud) condition $c4$. By the invariant (5)

$$\mathsf{ul}[(p1,o1)] + \mathsf{r2}[(p1,o1)] + \mathsf{w2}[(p1,o1)] + pr_{1,2}(\mathsf{cp})[(p1,o1)] + \mathsf{cr}[p1] + \mathsf{s}[p1] = 1$$

we know that there is a path from $c4$ to $c10$ on this invariant. As shown above, there cannot be a LdCnt(p1) event on this path. Remember, that there are no LdCnt(p1) events between $e1'$ and $e9$. An automaton that characterizes all possible paths to the first occurrence of an ul.(p1,o1) condition is shown in Fig. 106. There is only one event on this path: a UpdObj(o1,p1) event. This event $e7$ is the event required on the right hand side of requirement *B.2.2.1.* Since we know that $e8$ happens causally between $e1'$ and $e9$, we know that there is no Lck(o1,.) event between $e1'$ and $e8$. Thus, the LogDBu(p1) event $e8$ is the event required on the right hand side of *A.1.2.2.1.*

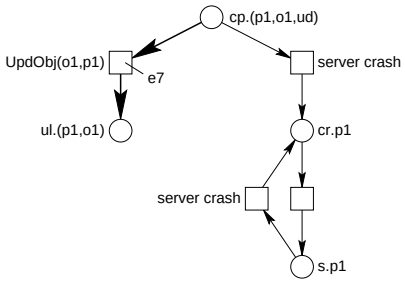


Figure 106: Path without a LdCnt(p1) event from a cp.(p1,o1,ud) to an ul.(p1,o1) condition along invariant (5)

Case ii, $e9$ is a t7 event with $p = p1$: In this case, the lock of $p1$ on $o1$ is released after the redo procedure for process $p1$. Fig. 107 shows the chain of causes for the t7 event with $p = p1$. There are three different cases:

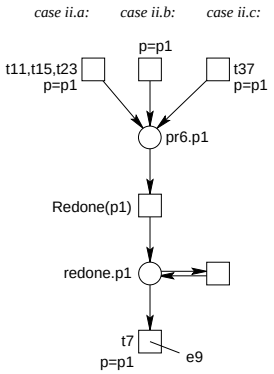


Figure 107: Chain of causes of t7 with $p = p1$

***case ii.a*:** The redo procedure does not update the database because the first log record encountered is an update record, a continuation record, or a DBu record, which corresponds to an occurrence of one of the following transitions t11, t15, t23 with $p = p1$.

***case ii.b*:** The redo procedure updates the database, and it terminates by writing a DBu record to the log file. This corresponds to the occurrence of transition t17 with $p = p1$.

***case ii.c*:** The redo procedure does not encounter a log record for process $p1$ at all. This corresponds to an occurrence of transition t37 with $p = p1$.

These cases are analog to the cases *i–iii* in the proof of requirement *A.1.2.2.1.* In *case ii.a* and *case ii.c*, we can use exactly the same argument. Therefore, we start with these two cases and deal with *case ii.b* at the end.

***Case ii.a*:** By the same argument as in *case i* of the proof of *A.1.2.2.1*, the occurrence of transitions t11, t15, and t23 with $p = p1$ results from a LogObj(.,p1), a LogCnt(p1), or a LogDBu(p1) record event $e10$ that causally occurred after the commit event $e1'$ (see p. 65 and Fig. 73, 74, and 76). From our assumptions and the construction of $e10$, we know that no LdCnt(p1) event occurs between $e1'$ and $e10$.

Figure 108 shows all possible paths of causalities from $e1'$ to $e10$ along invariant (1); remember that LdCnt(p1) events do not occur on these paths. The automaton shows that there exists a LogDBu(p1) event $e8$ on this path. Since we know already that there are no Lck(o1) events between $e1'$ and $e9$, and since $e8$ occurs between $e1'$ and $e9$, event $e8$ meets the requirements on the right hand side of *B.2.2.1.*

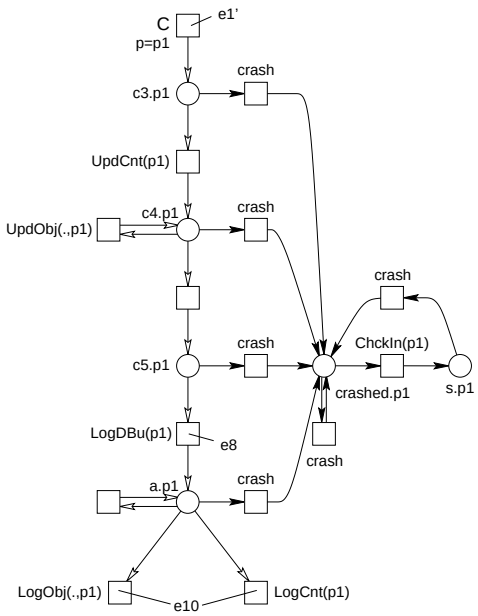


Figure 108: Automaton proving the existence of the LogDBu(p1) event

It remains to show that there exists a UpdObj(o1,p1) event $e7$ between $e1'$ and $e8$, and a path of data causality from $e1$ to $e7$: From the Petri net model, we know that event $e8$ has a ul.(p1,o1) condition $c10$ in its preset. By invariant (5), this condition cannot occur concurrently to the cp.(p1,o1,ud) condition $c4$. Since $e1'$ happens before $e8$, we know that $c4$ occurs also before $c10$. Figure 106 shows that on each path from $c4$ to $c10$ an UpdObj(o1,p1) event occurs, and that there is a path of data causality from $e1$ via $c4$ to event $e7$.

***Case ii.c*:** By the same arguments as in *case iii* in the proof of *A.1.2.2.1*, we can show that this case is impossible. There is at least one log record for process $p1$ which that must be encountered during the redo procedure for process $p1$ (see *case iii* of proof of *A.1.2.2.1* on p. 68).

***Case ii.b*:** In this case, we know that some updates where made during the redo procedure for process $p1$. It remains to show that also the effect of $e1$ was updated to the database—either during the redo procedure or before.

Figure 109 shows the chain of causes of the t17 event $e10$ with $p = p1$. Since in this automaton, we have only places of place invariant (2), and since k.p1 does not occur in this automaton, we know that the initial Ignore(p1) event of this automaton occurs causally after the commit event $e1'$. By Property 5, we know $m \geq k$. Let m' be the sequence number of the continuation record encountered by the t13 event preceding

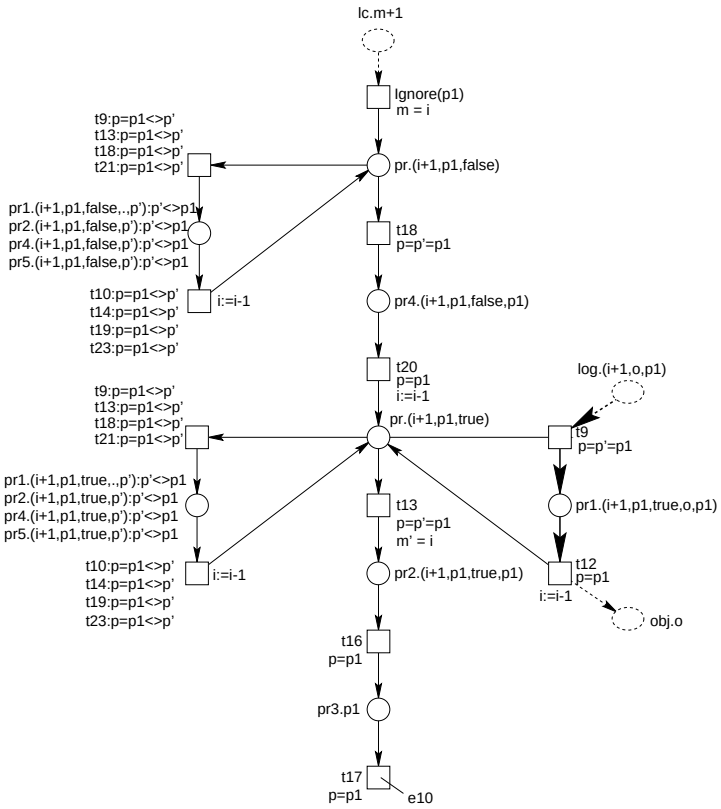


Figure 109: Chain of causes of t17 with $p = p1$

t17 event $e10$. We distinguish two cases:

$m' = j$ The continuation record is the one written by the LogCnt(p1) event $e6$. By $j < l < k \leq m$, we know by the structure of the automaton, that a log record with sequence number $l + 1$ is read by an event on some path of the automaton. Since sequence numbers of log records are unique (Property 6), we know that this is the log record written by LogObj(o1,p1) event $e1$ and there is a path of data causality from $e1$ to this read event. The only event of the automaton that can read this log record is the t9 event on the right hand side of the automaton (with $p = p' = p1$). Then, we have also a t12 event, which updates the object to the database. Obviously, there is a path of data causality from $e1$ via t9 to t12. So, the occurrence of the update event t12 is the event $e7$ required on the right hand side of *B.2.2.1.* Moreover, the t17 event $e10$ is the event $e8$ required on the right hand side of *B.2.2.1.* By the arguments given before, we know that there are no intermediate Lck(o1,.) events.

$m' > j$ The continuation record is one which is written by a LogCnt(p1) event $e11$ that happens causally after $e6$. Since we know that there is no LogCnt(p1) event between $e6$ and $e1'$, it must be causally after the commit event $e1'$. Thus, we have a LogCnt(p1) event $e11$ that occurs causally after $e1'$ without intermediate LdCnt(p1) events. This case has already been considered in *case i.a.*

Case iii, $e9$ is a t24 event: At last, we consider the case that the event $e9$ is the successful termination of a restart of the server. First, we show that the log records written by events $e6$, $e1$ and $e1'$ are read during the

restart procedure of the server. Figure 110 shows the chain of causes for the t24 event $e9$. Since the automaton contains no place k and by the place invariant (2) $\mathsf{k}[p1]+\mathsf{uk}[p1]+\mathsf{rel}[p1]+\mathsf{pri}[p1]+\mathsf{red}[p1]+\mid \mathsf{sri}+\mathsf{svcr} \mid = 1$, we know that the t36 event initiating the restart of the server occurs causally after the commit event $e1'$. Thus, we have $m \geq k+1$ by Property 5. By the structure of the automaton (alternation of events reading a log record and decreasing i) and by the uniqueness of the sequence numbers of log records (Property 6), the log records $(k+1, cmt, p1)$, $(l+1, o1, p1)$, and $(j+1, cnt, p1)$ are read on the path form t36 to t24. Altogether,

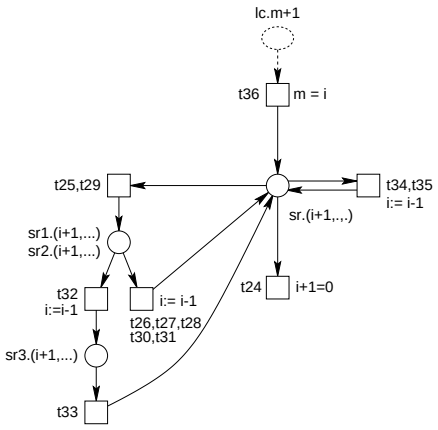


Figure 110: The automaton for the restart of the server

we have the situation shown in Fig. 111, and the t36 event occurs causally after the commit event $e1'$. By a

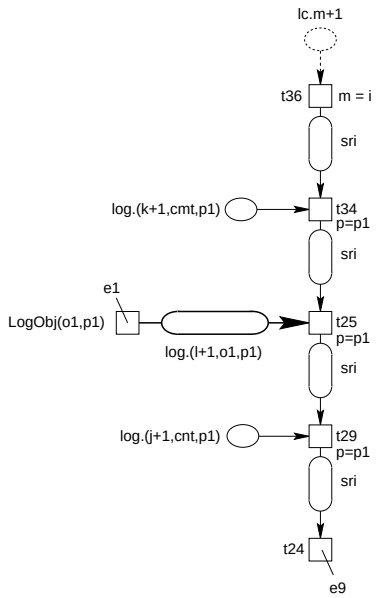


Figure 111: Situation during the restart of the server

simple automaton and the uniqueness of sequence numbers of log records, we know that there is a path of data causality from the LogObj(o1,p1) event $e1$ to the t25 event.

In the following, we will show that after the t25 event, there is an event updating the update record to the database. In the end, we will show that we also have an event $e8$ that writes a DBu record to the log file after the update and before event $e9$.

Let us have a closer look to the t25 events that reads the $(l+1, o1, p1)$ record. Figure 112 shows the three possible events that may succeed the t25 event. Note that one of these events must exist because the restart of the server was terminated successfully (technically, the only possible paths from the t25 event to t24 event is via these events).

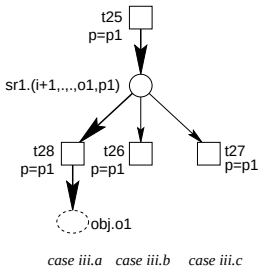


Figure 112: Possible actions upon read update record

***Case iii.a*:** The update record is updated to the database by the t28 event. This is the update event $e7$ required on the right hand side of *B.2.2.1*. In this case, it remains to show that we also have the event $e8$ that writes the DBu record after $e7$ and before $e9$ (see below).

***Case iii.b*:** The update record is not updated to the database because object $o1$ is already marked as redone; i.e. we have sr1.(l+1,[..o1..],..,o1,p1). We will show below that this case is impossible. The informal idea is that, in this case, there was an intermediate lock operation on object $o1$; this, however, is impossible because the lock of $p1$ was not yet released.

***Case iii.c*:** The update record is not updated to the database because process $p1$ is not marked as committed in the currently scanned part of the log file; i.e. we have sr1.(l+1,..,c,o1,p1) with $c[p1] = 0$. We will show below that also this case is impossible. The informal idea is that, in this case, there must be another LogCnt(p1) event between $e6$ and $e1$—which is a contradiction to the considered situation.

Next, we will prove that *case iii.b* and *case iii.c* are impossible. Then, we will continue *case iii.a* by showing the existence of event $e8$ writing the checkpoint record.

Impossibility of *case iii.b*: If we have sr1(j+1,[..o1..],..,o1,p1), we know that there must have been an event t28 that updates object $o1$ before. Figure 113 shows the chain of causes for this condition. There must be an intermediate t28 event with $o = o1$ that updates a log record $(l'+1, o1, p2)$ to the database, where $l' > l$. By a simple automaton we know that there must be a corresponding LogObj(o1,p2) event $e11$. By *B.3.2*, we know that $e11$ is a directly committed event. By Property 10, we have the situation shown in Fig. 114. Let us investigate when these events $e13$, $e12$, $e11$, and $e14$ could occur (in relation to the events of the situation from Fig. 102: Certainly $e11$ occurs causally before $e9$; by $l' > l$ we know that $e11$ is different form $e1$. By Property 5, $e11$ occurs causally after $e1$. By Property 8, the lock events $e4$ and $e13$ are also different. Thus, by Requirement *B.1.3*, we either have $e4$ occurs causally after $e14$ (which establishes a cyclic path of causalities and thus is impossible) or $e13$ occurs causally after $e1$. In the latter case, we have a lck.(o1,[]) condition between $e1'$ and $e9$. This, however, was impossible in the situation of Fig. 102 ($c8$ is the first occurrence of a lck.(o1,.) condition) after $e4$. Altogether, we know that *case iii.b* is impossible.

Impossibility of *case iii.c*: Let us consider the situation of Fig. 111 and Fig. 112 again. If the t27 event occurs, we have a sr1.(j+1,..,c,o1,p1) condition before with $c[p1] = 0$. On the other hand, we know that after the occurrence of the t34 event, we have a sr.(k,..,[..p1..]) condition (when a commit record for $p1$ is encountered, $p1$ is added to the list of committed events). By $k \geq j+1$, we know that on the path from the t34 to t27, $p1$ must be removed from the list, which is possible only when a continuation record is encountered. The corresponding automaton is shown in Fig. 115. We know that the t29 event with $p = p1$

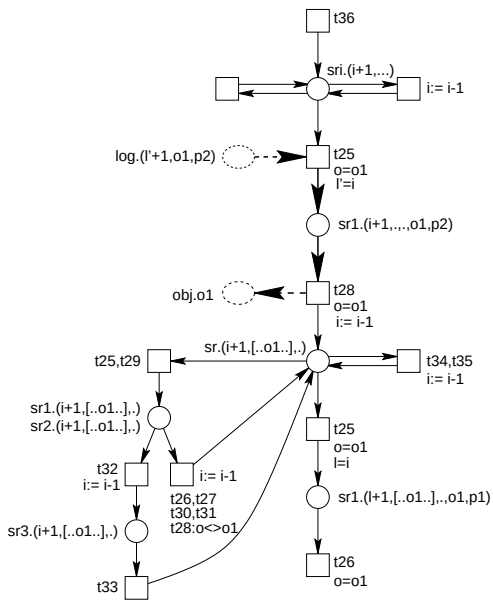


Figure 113: Chain of causes for sr1(j+1,[..o1..],.,o1,p1)

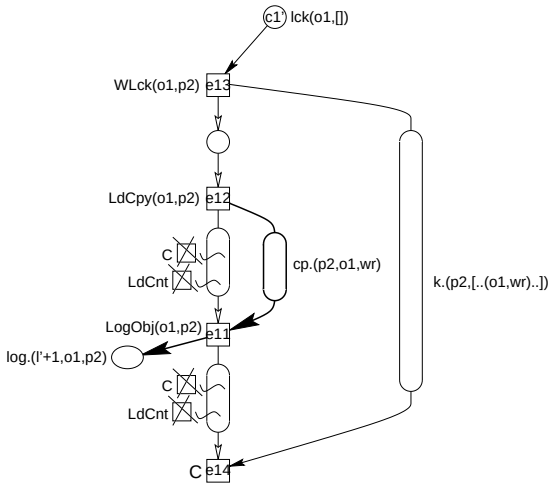


Figure 114: The context of the other LogObj(o1,p2) event

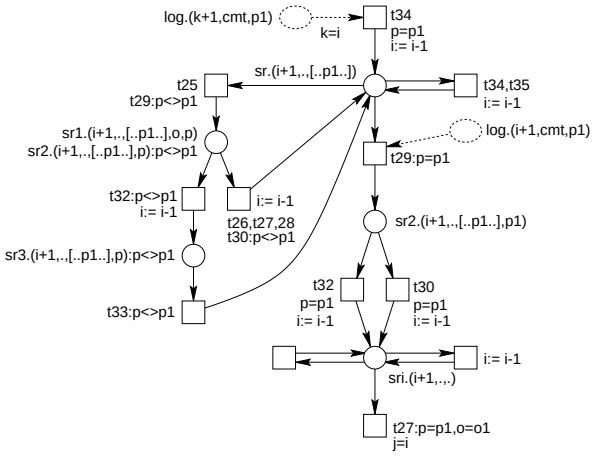


Figure 115: Not in commit list implies intermediate log continuation record

reads a continuation record $(i+1, cnt, p1)$ with $j < i < k$. By Property 6.1, there is be a LogCnt(p1) event that writes this log record. By Property 5, this LogCnt(p1) occurs causally between the LogObj(o1,p1) event $e1$ and the commit event $e1'$. By Properties 2, 3, and invariant (1), we know that the LogCnt(p1) event must occur on the path of program causality from $e1$ to $e1'$. This is impossible in this situation because the only events that occur on this path are LogObj events by assumption.

Existence of an event writing a DBu record for $p1$: Since *case iii.b* and *case iii.c* turned out to be impossible, it is sufficient to consider *case iii.a* in the following. It remains to show that there exists an event $e8$ that writes a DBu record after the update event $e7$ and before $e9$.

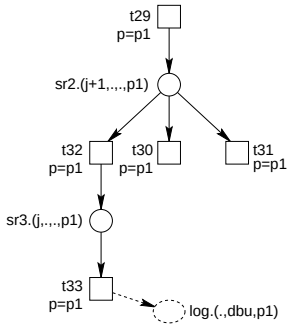


Figure 116: Reactions to a read continuation record

It will turn out that this event is an reaction to a read continuation record of process $p1$. So let us consider the t29 event of Fig. 111 in more detail. The possible reactions to a read continuation record are shown in the automaton from Fig. 116. Again, there are three possibilities:

Case iii.d The continuation is updated to the database, and then a t33 event writes the DBu record for $p1$. This event occurs causally after the update event t25 and causally before $e9$. Thus, in this case, we have finished the proof of requirement *B.2.2.1.*

Case iii.e The continuation record is not updated to the database and there is no DBu record written because the continuation is already in the redone list. In that case, a t30 event occurs with a condition sr2.(j+1,[..p1..],.,p1) in its preset.

We will show below that, in this case, we have another LogCnt(p1) causally after the commit event $e1'$. By arguments already given in *case ii.a*, we know that, in this case, the situation meets the right hand side of *B.2.2.1*.

Case iii.f The continuation record is not updated to the database and there is no DBu record written because process $p1$ is not in the commit list. In that case, a t31 event occurs with a condition sr2.(j+1,..,c,p1) with $c[p1] = 0$ in its preset.

Since we have a t34 event with $p = p1$ that encountered the commit record $(k+1, cmt, p1)$ before, we know that there must be an intermediate event that reads a continuation record for process $p1$. We can use the same automaton as in *case iii.c* (see Fig. 115). By the same arguments as in *case iii.c*, there must be another LogCnt(p1) event on the path of program causality from the LogCnt(p1) $e6$ to the commit event $e1'$. This, however, was excluded by assumption. Thus, *case iii.f* is impossible.

It remains to show that, in *case iii.e*, there is a LogCnt(p1) event that occurs causally after the commit event $e1'$ and before $e9$. By the arguments of *case ii.a*, this proves requirement *B.2.2.1*.

In order to prove the existence of this LogCnt(p1) event, let us investigate the chain of causes of the t29 event with $p = p1$. The automaton is shown in Fig. 117 (note that this automaton is similar to the one of Fig.113 in *case iii.b*). Thus, we know that there is a log record $(j'+1, cnt, p1)$ with $j < j'$. By

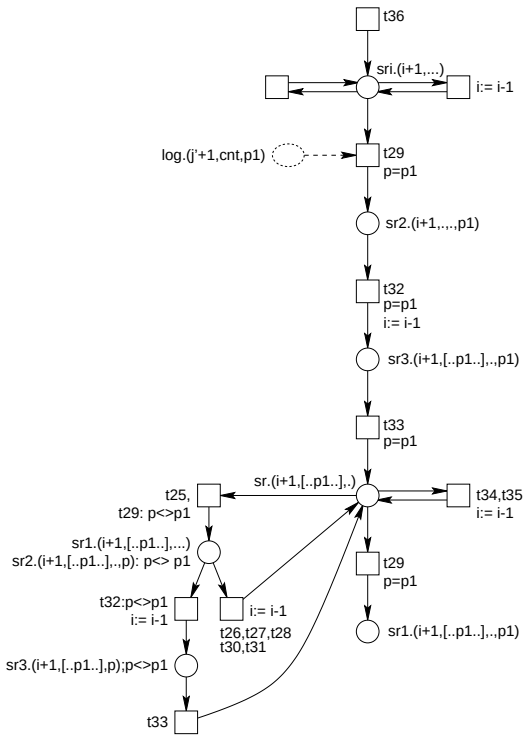


Figure 117: Chain of causes for sr1(j+1,[..p1..],..,o1,p1)

Property 6, we know that there is a corresponding LogCnt(p1) event. By Property 5, this LogCnt(p1) event occurs causally after $e6$. Since we know that no LogCnt(p1) event occurs between $e6$ and $e1'$, we also know that the LogCnt(p1) event occurs causally after $e1'$. Since the log record is read before $e9$, the LogCnt(p1) event also occurs causally before $e9$. This finishes our proof of requirement *B.2.2.1*

7.15 B.2.2.2: No skip of updates

At last, we verify requirement *B.2.2.2*, which is graphically represented in Fig. 118 again. Note that we have renamed the commit event $e1'$ to $e2$ for convenience. We must show that, after an object is updated in the database by event $e7$, another update event $e9$ does not write an older value to the database.

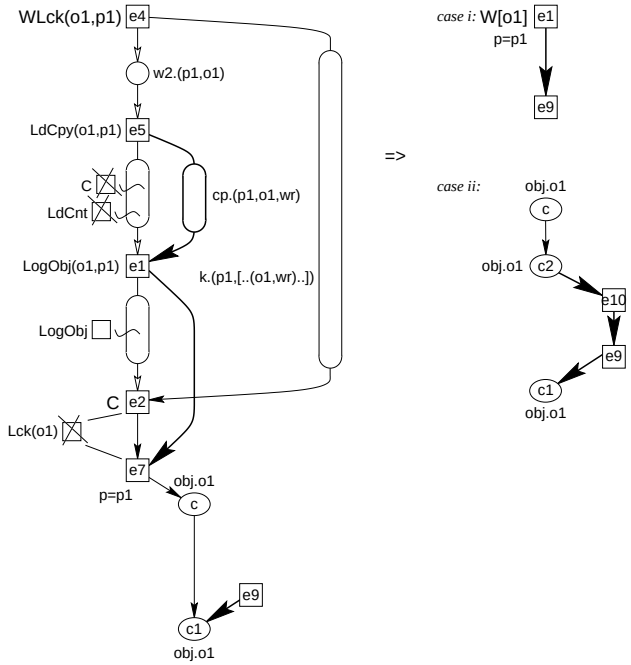


Figure 118: Requirement *B.2.2.2*

In the Petri net model, there are only four possible transitions with a data causality arc to place object. Thus, event $e9$ corresponds to one of the following transitions in the Petri net model:

***case i*:** transition LdCpy(o1,p2) for some process $p2$.

***case ii*:** transition UpdObj(o1,p2) for some process $p2$

***case iii*:** transition t12, which updates object $o1$ during the redo procedure for some process $p2$.

***case iv*:** transition t28, which updates object $o1$ during a restart of the server.

The different cases for event $e9$ are shown in Fig. 119. Note that c' occurs causally after c because of invariant (6) and because $e9$ happens causally after c— otherwise conditions c and c' are concurrent, which contradicts invariant (): $\mathsf{obj}[o1] = 1$.

In the following we consider each case separately.

Case i If we choose $c2 = c'$ and $e10 = e9$, *case i* immediately meets the situation on the right hand side of requirement *B.2.2.2* (*case ii*).

Case ii In case $e9$ is an UpdObj(o1,p2) event, we first consider a chain of causes of the update event $e9$. An automaton for a chain of causes of an UpdObj(o1,p2) event is shown in Fig. 120: the life cycle of a copy of an object starting with a lock event. This gives us the situation shown in Fig. 121, where we have added a chain of k.(p2,[..(o1,wr)..]) conditions from event $e1'$ to event $e9$. The argument for this chain is the following: First, we know that the WLck(o1,p2) event has a condition k.(p2,[..(o1,wr)..]) in its postset. Moreover, the

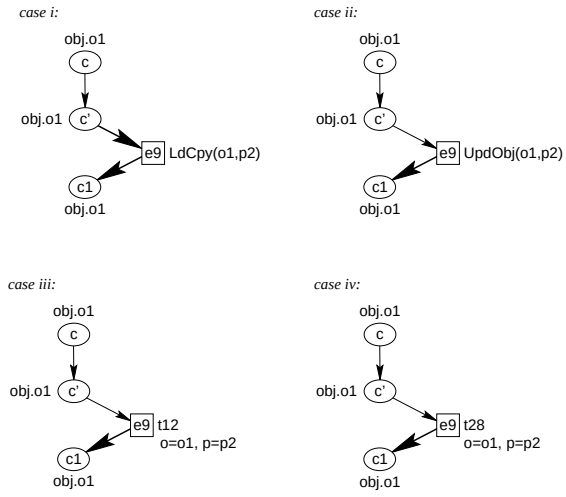


Figure 119: Requirement *B.2.2.2*: Cases for event *e*9

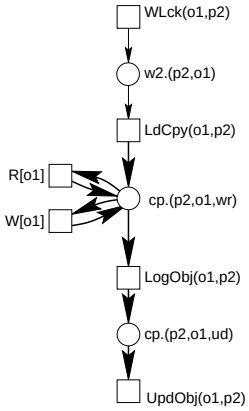


Figure 120: Chain of causes for UpdObj(o1,p2) event

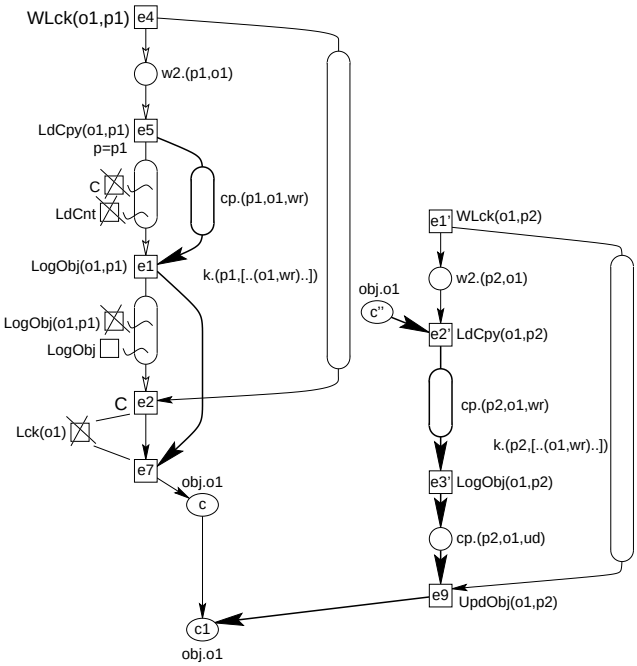


Figure 121: Situation in *case ii*

UpdObj(o1,p2) $e9$ has a condition k.p2 in its preset. By Property 9.1, there are no LdCnt(p1) and RelLck(p1) events between $e1'$ and $e9$. By Property 9.2, there is a chain of k.(p2,[..(o1,wr)..]) conditions between $e1'$ and $e9$.

Next, we distinguish two cases: $e3' = e1$ and $e3' \neq e1$. If $e3' = e1$, we know that we have a path of data causality from $e1$ to $e9$. Thus, the situation meets the right hand side of requirement *B.2.2.2* (*case i*). If $e3' \neq e1$, we know, by Property 8, that the corresponding lock events $e1'$ and $e4$ are also different. By requirement *B.1.3*, we know that either $e9$ occurs causally before $e4$ (this is impossible since this induces a cyclic causality) or $e1'$ occurs causally after $e2$. Moreover, we know by assumption that $e1'$ must occur after $e7$. Therefore, $e2'$ occurs causally after $e7$. With $c2 = c''$ and $e10 = e2'$, this situation meets the right hand side of requirement *B.2.2.2* (*case ii*). Note that c'' occur causally after c because of invariant (6).

Case iii Now, we consider the case that $e9$ is a t12 event with $p = p2$ and $o = o1$. Remember that $p2$ could be equal to $p1$. The automaton from Fig. 122 shows the chain of causes of this event. Basically, this is the redo procedure for process $p2$: The redo procedure is initiated by the Ignore(p2) event $e5'$; during the redo procedure, a commit record of process $p2$ (event $e6'$) and an update record of process $p2$ for object $o1$ (event $e7'$) are encountered. These log records are written by the corresponding log events) $e3'$ and $e4'$. This situation is shown in Fig. 123. From the automaton, we know $k < l \leq m$. Moreover, we know that during the redo procedure between event $e5'$ and $e6'$ there is a log record for each $i + 1$ with $k \leq i < m$ and the encountered log records are different from $(i + 1, dbu, p2)$; the encountered log records for i with $k \leq i \leq l$ are different from $(i + 1, cnt, p2)$. By the uniqueness of log records (Property 6), we know that there are no log records $(i + 1, dbu, p2)$ with $k \leq i < m$ and no log records $(i + 1, cnt, p2)$ with $k \leq i \leq l$. Thus, we know that there is no LogCnt(p1) event between $e3'$ and $e4'$ by Property 5. By Property 1, we know that there is a path of program causality from the LogObj(o1,p2) event $e3'$ to the commit event $e4'$ on which neither a LdCnt event nor a C event occurs.

Thus, $e3'$ is a directly committed LogObj(o1,p2) event, which allows us to apply Property 10. This gives us the situation from Fig. 124.

Now, we distinguish two cases: the LogObj events $e1$ and $e3'$ coincide or are different. For $e1 = e3'$, we

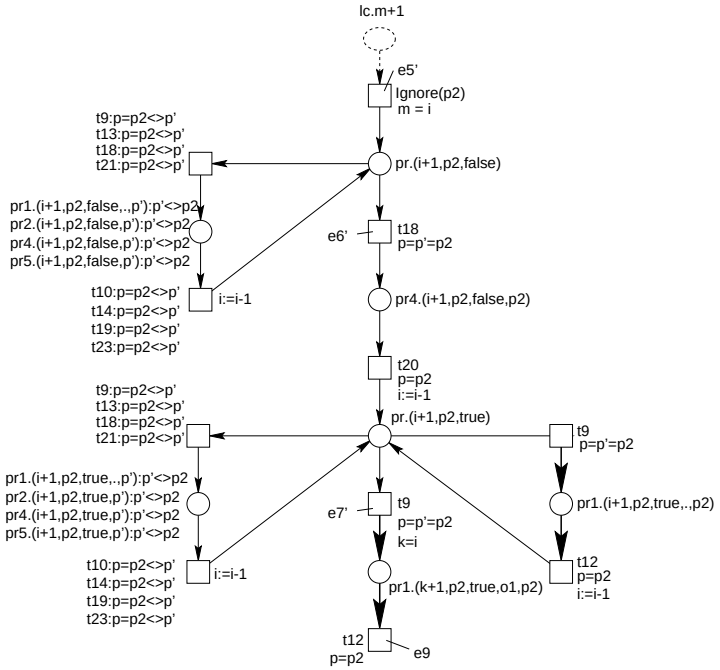


Figure 122: Chain of causes for t12 event $e9$

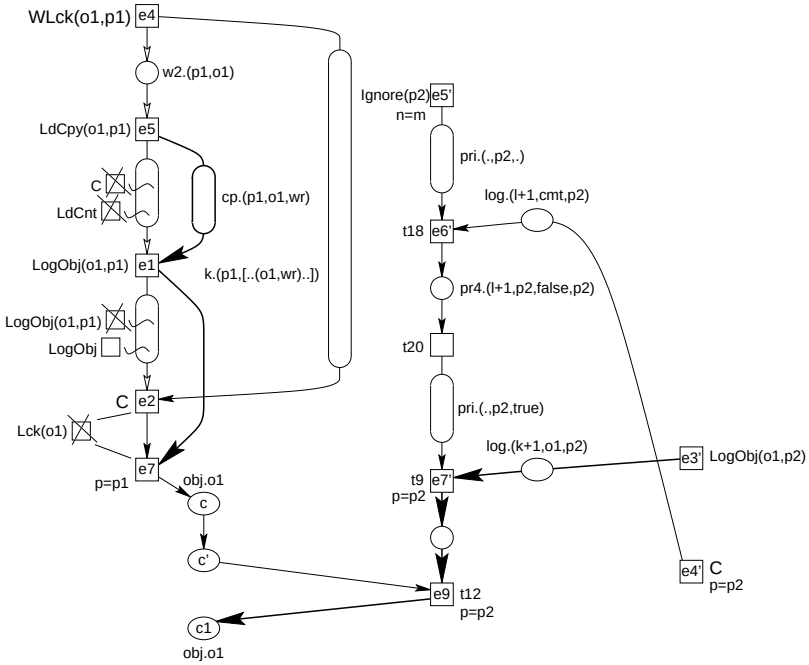


Figure 123: The redo procedure initiating the t12 event

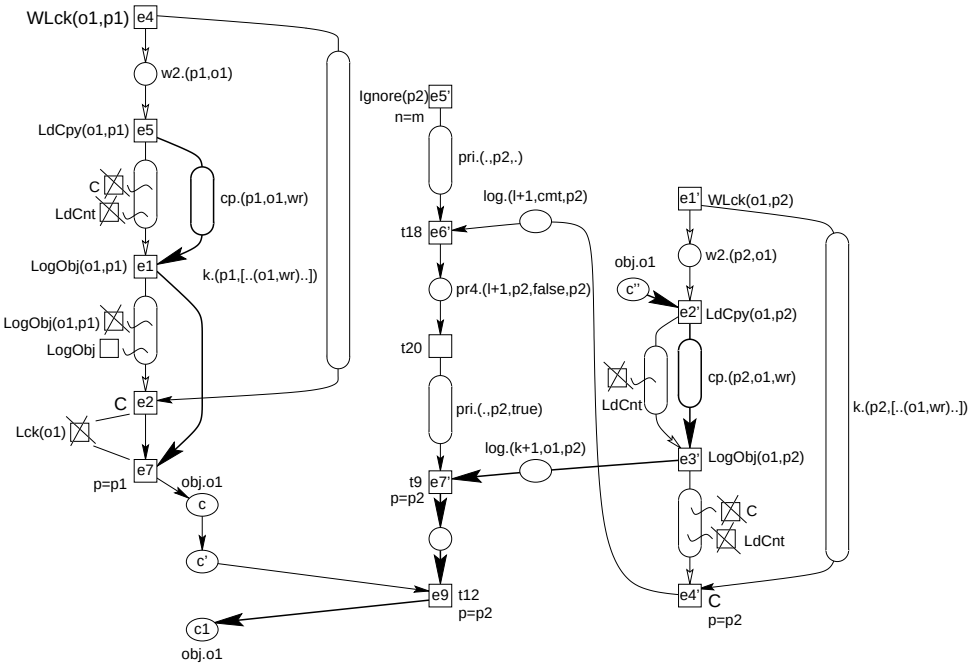


Figure 124: The corresponding LdCpy(o1,p2) and WLck(o1,p2) events

have a path of data causality from $e1$ via $e7'$ to $e9$. Thus, we have the situation of the right hand side of requirement *B.2.2.2* (*case i*).

For $e1 \neq e3'$ we know, by Property 8, that $e4 \neq e1'$. By requirement *B.1.2*, we know that either $e2$ happens causally before $e1'$ or we know that $e4'$ happens causally before $e4$. For $e2 < e1'$, we also know that $e1'$ happens causally after $e7$ because, by assumption, no lock event occur between $e2$ and $e7$. Thus the situation meets the right hand side of requirement *B.2.2.2* (*case ii*) with $c2 = c''$ and $e10 = e2'$.

For $e4' < e4$, we show that requirement *B.2.2.1* is violated: The commit event $e4'$ occurs causally before the WLck(o1,p1) event $e4$. We will show that there is no DBu record of process $p2$ written between $e4'$ and $e4$—the existence of this DBu record, however, is required by *B.2.2.1*. Let us assume that such a DBu record exists: This DBu record for $p2$ could only be written by a t17, a t33, or a LogDBu(p2) event. By invariant (2) and the structure of these transitions, this event cannot occur between events $e5'$ and $e9$. Thus, the event writing the DBu record for $p2$ must occur between $e4'$ and $e5'$. Thus, it receives a sequence number $i+1$ with $k \leq i \leq l$. From the automaton from Fig. 122, however, we know that there are no $(i+1, dbu, p2)$ records with $k \leq i \leq l$. Thus $e4' < e4$ is impossible, which finishes the proof of *case iii*.

Case iv At last, we consider the case of an update of object $o1$ during the restart of the server (transition t28). Figure 125 shows a backward automaton for the chain of causes for transition t28. The resulting situation is shown in Fig. 126. Note that, for all conditions on the chain from $e5'$ to $e9$, we have $r[o1] = 0$, i.e. object $o1$ is not updated during the backwards scan before event $e9$. This property will be exploited at the end of this proof. In addition to the chain of causes derived from the above automaton, we have indicated also the LogObj(o1,p2) event $e3'$ that writes the update record encountered by event $e7'$. By requirement *B.3.2*, we know that there is a direct commit event $e4'$ for this LogObj(o1,p2) event $e3'$. By Property 10, we get the situation from Fig. 127.

Let us consider the situation of Fig. 127 and distinguish different cases: First, let us assume $e1 = e3'$. In this case, we have a data causality from $e1$ to $e9$; thus, the situation meets the right hand side of requirement *B.2.2.2* (*case i*).

For the rest of the proof, let us assume $e1 \neq e3'$. By Property 8, $e1 \neq e3'$ implies $e4 \neq e1'$. By

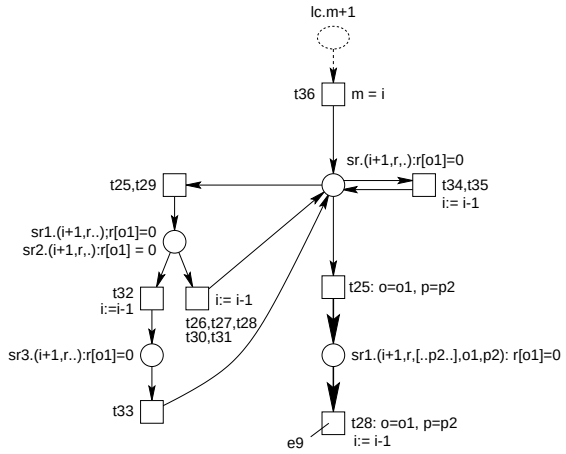


Figure 125: Chain of causes for t28 event

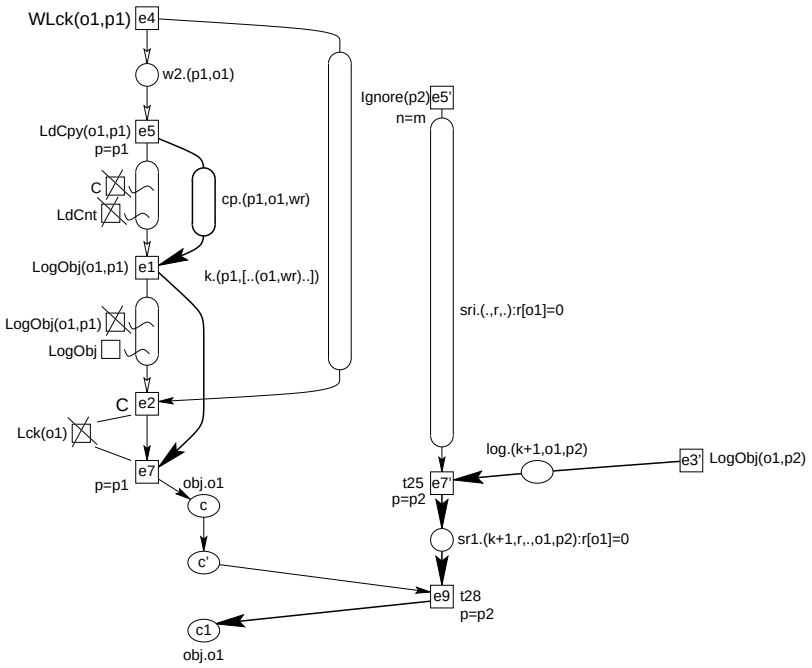


Figure 126: Backwards scan during restart of the server

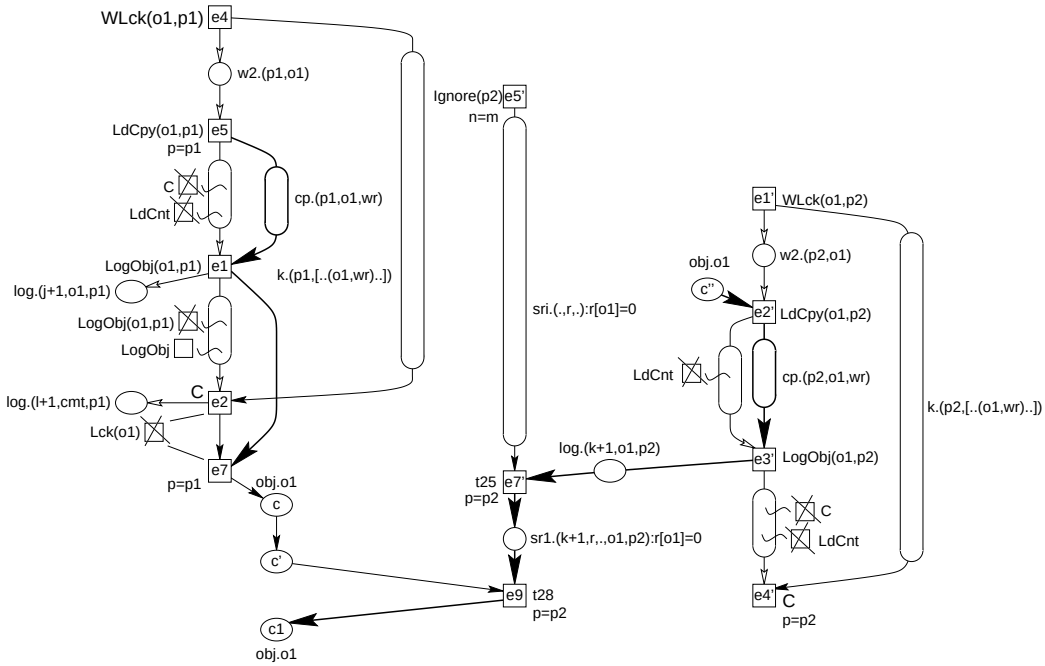


Figure 127: Complete situation in *case iv*

requirement *B.1.3*, this implies: $e2 \leq e1'$ or $e4' \leq e4$. We first consider the case $e2 \leq e1'$: By assumption, we know that no Lck(o1,.) event occurs between $e2$ and $e7$. Thus, $e1'$ occurs causally after $e7$. Thus, $e2'$ occurs causally after $e7$ which, by invariant (6) also implies $c \leq c''$. With $c2 = c''$ and $e10 = e2'$ the situation meets the right hand side of requirement *B.2.2.2* (case *case ii*).

It remains to consider the case $e4' \leq e4$. We will show that this case is impossible: Let us consider the transitions that could correspond to event $e7$. This could be transition LogObj(o1,p1), transition t12 with $p = p1$ and $o = o1$, or transition t28 with $p = p1$ or $o = o1$. Each of these events has a condition k.p1, pri.p1, or sri in its immediate context. By invariant (2), this implies that $e7$ occurs causally before $e5'$ or occurs on the path of sri conditions from $e5'$ to $e6'$. If event $e7$ occurs on the path from $e5'$ to $e6'$, the event $e7$ is a t28 event with $p = p1$ and $o = o1$. This however is impossible, because this event adds object $o1$ to the redo list—remember that all conditions on the chain from $e5'$ to $e7'$ are sri.(.,r,.) with $r[o1] = 0$. This leaves the case $e7 \leq e5'$. In that case, we have $e3' \leq e1 \leq e2$ and, by Property 5, $k < j < l < m$. By the automaton of Fig. 125 and the uniqueness of sequence numbers of log records (Property 6), we know that the log records written by $e1$ and $e2$ are read on the path from $e6'$ to $e7'$ as shown in Fig. 128. Remember that, on the path from $e5'$ to $e7'$, we have only conditions sri.(.,r,.) with $r[o1] = 0$. Thus event $e9'$ can only be a t27 event. This, however, implies that $c[p] = 0$ (i.e. process $p1$ is no longer marked as committed, when the update record for $o1$ of process $p1$ is encountered). On the other hand, we have a sr.(.,.,[..p1..]) after the t34 event $e6'$. The automaton from Fig. 129 shows that, on the path from $e6'$ to $e9'$, a continuation record $(i + 1, cnt, p1)$ with $j \leq i \leq l$ is encountered ($p1$ is removed from the commit list by t32). This implies, that there is a LogCnt(p1) event causally between $e1$ and $e2$. By invariant (1) and the context of transition LogCnt(p1) this event must occur on the path of program causality form $e1$ to $e2$. By assumption, we know that only LogObj events can occur on this path; thus this last case is also impossible.

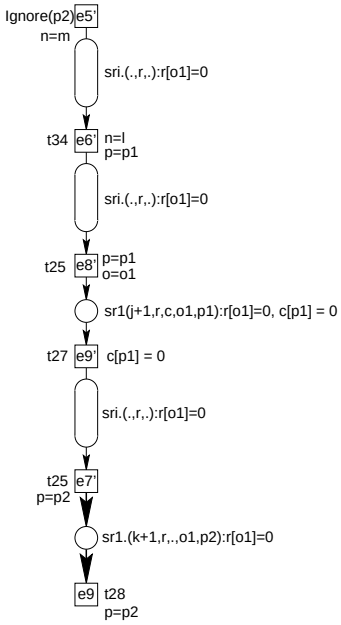


Figure 128: More details of the restart procedure

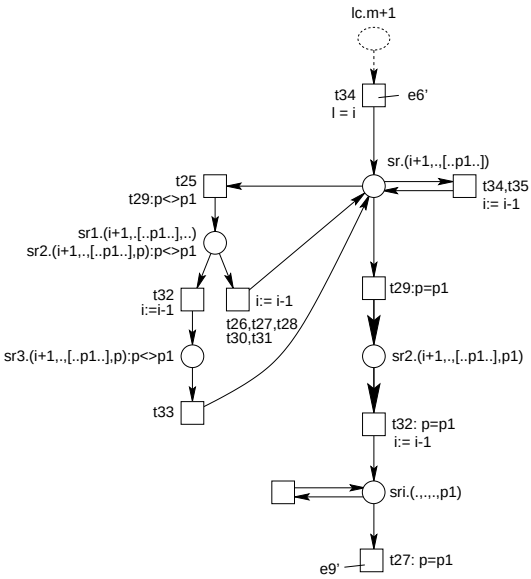


Figure 129: Automaton that proves the existence of an intermediate LogCnt(p1) event.

8 Basic properties

In this section, we summarize the definitions of all abbreviations, and we prove all basic properties, used in the previous proofs.

8.1 Abbreviations

We start with the abbreviations. Table 3 shows the abbreviations for place names.

cn	=	continuations
cp	=	copies
cr	=	crashed processes
kn	=	known
lc	=	log counter
lck	=	locks
ob	=	objects
pr	=	process redo
p	=	processes
sr	=	server restart
svcr	=	server crash
uk	=	unknown
unl	=	unlocked
wlc	=	wlckcnt = wlckcount
xlck	=	xlocks

Table 3: Abbreviations of place names

Table 4 shows the definition of collections of places. Each collection is represented as a sum of places. Basically, a collection sums up all tokens in different places. We use the projection to the first or second component in order to select a particular component of a token in the sum. The projection functions are denoted by pr_1 and pr_2, respectively.

a	=	$\mathsf{p} + pr_1(\mathsf{r1} + \mathsf{r2} + \mathsf{w1} + \mathsf{w2}) + \mathsf{c1} + \mathsf{c2} + \mathsf{c3} + \mathsf{c4} + \mathsf{c5} + \mathsf{c6}$
a'	=	$\mathsf{p} + pr_1(\mathsf{r1} + \mathsf{r2} + \mathsf{w1} + \mathsf{w2}) + \mathsf{c1} + \mathsf{c4} + \mathsf{c5} + \mathsf{c6}$
cnt	=	$pr_1(\mathsf{continuations})$
k	=	$pr_1(\mathsf{known})$
obj	=	$pr_1(\mathsf{objects})$
pri	=	$pr_2(\mathsf{pr} + \mathsf{pr1} + \mathsf{pr2} + \mathsf{pr3} + \mathsf{pr4} + \mathsf{pr5} + \mathsf{pr6})$
red	=	$pr_1(\mathsf{redone})$
rel	=	$pr_1(\mathsf{releasing})$
sri	=	$\mathsf{sr} + \mathsf{sr1} + \mathsf{sr2} + \mathsf{sr3}$

Table 4: Abbreviations of projections and place collections

8.2 Invariants

Next, we define the invariants that are used in the proof. The invariants are shown in Table 5. All invariants are an immediate consequence of a so-called *place invariant* of a Petri net [18, 19]. Invariants (1)–(6) and (10) can be immediately checked from the Petri net model: Each transition that removes one token from the corresponding collection of places also adds one token to the corresponding collection. Initially, the number of considered tokens is 1. Therefore, the sum of considered tokens in the collection is 1 throughout

$$
\begin{array}{ll}
(1) & \mathsf{a}[p1] + \mathsf{cr}[p1] + \mathsf{s}[p1] = 1 \\
(2) & \mathsf{k}[p1] + \mathsf{uk}[p1] + \mathsf{rel}[p1] + \mathsf{pri}[p1] + \mathsf{red}[p1] + \mid \mathsf{sri} + \mathsf{svcr} \mid = 1 \\
(3) & \mathsf{cnt}[p1] = 1 \\
(4) & pr_1(\mathsf{xlck} + \mathsf{lck})[\mathsf{o1}] + \mid \mathsf{svcr} + \mathsf{sri} \mid = 1 \\
(5) & \mathsf{ul}[(p1, o1)] + \mathsf{r2}[(p1, o1)] + \mathsf{w2}[(p1, o1)] + pr_{1,2}(\mathsf{cp})[(p1, o1)] + \mathsf{cr}[p1] + \mathsf{s}[p1] = 1 \\
(6) & \mathsf{obj}[o1] = 1 \\
(7) & \mathsf{wlckcnt}(p1, 0) \Rightarrow \neg \mathsf{cp}(p1, o1, wr) \\
(8) & \mathsf{locks}(o1, []) \Rightarrow \forall \mathsf{p}, \forall m' : (\mathsf{k}.(\mathsf{p}, m') \Rightarrow m'[(o1, rd)] = 0) \\
(9) & \mathsf{locks}(o1, m) \Rightarrow \forall \mathsf{p}, \forall m', \forall x : (\mathsf{k}.(\mathsf{p}, m') \Rightarrow m'[(o1, x)] = 0) \\
(10) & \mid lc \mid = 1
\end{array}
$$

Table 5: Invariants

the execution. This can be easily checked for each transition of the Petri net model separately—there are techniques to check this fully automatically [3]. Therefore, we skip the details of these proofs. We only mention the non-standard transitions of the Petri net model: the transition that models the crash of a client and the transition that models the crash of the server. These transitions remove all tokens from a particular set of places. For each of invariant (1)–(6) and (10), the occurrence of these transitions establishes the initial marking of all places of the corresponding invariant.

Invariants (7), (8), and (9) are slightly more involved. They are an immediate consequence of the invariants below, which are place invariants and can be checked by standard techniques:

$$
\begin{array}{ll}
(7') & \sum_n \mathsf{wlckcnt}[(p1, n)] \cdot n = \sum_o \mathsf{cp}[(p1, o, wr)] + \mathsf{w2}[(p1, o)] \\
(8') & \sum_p \mathsf{xlocks}[(o1, o)] = \sum_{p,m} \mathsf{k}[(p, m)] \cdot m[(o1, wr)] \\
(9') & \mathsf{locks}[(o1, m)] \cdot \mid m \mid = \sum_{p,m} \mathsf{k}[(p, m)] \cdot m[(o1, rd)]
\end{array}
$$

Invariant (7′) says that the number of write locks for process $p1$, which are counted in $\mathsf{wlckcnt}(p1, n)$ is equal to the number of write copies of process $p1$ plus the number of copies that $p1$ is going to receive (i.e. when it is in state w2 of the write protocol). Invariants (8′) and (9′) says that the number of locks on object $o1$ that are represented in the corresponding lock lists (xlocks or locks) coincides with the number of locks on object $o1$ that are represented in the list of known processes (k). (8′) considers the number of read locks; (9′) considers the number of write locks.

These properties can be also verified for each transition separately. Note that the non-standard transitions modeling the crashes reset both sides of the equation to 0.

8.3 Behavioral properties

Next, we verify the behavioral properties.

Property 1 *Let $e1$ be a* LogCnt(p1) *or a* LogObj(o1,p1) *event, and let $e2$ be a commit event of process $p1$. Moreover, let there be no* LogCnt(p1) *event that occurs between $e1$ and $e2$. Then, there is a path of program causality from $e1$ to $e2$ on which only* c2.p1 *conditions occur and on which no* LdCnt *events,* RelLck *events, or* C *events occur.*

Figure 130 shows a graphical representation of Property 1. The proof of Property 1 is in two steps. First, event $e1$ has a c2.p1 condition in its postset, and $e2$ has a c2.p1 in its preset. By invariant (1), we know that there is a path of causality on which only conditions a.p1, cr.p1, and s.p1 occur (remember that, by definition of a, condition c2.p1 is a a.p1 condition). By assumption, we know that no LdCnt(p1) event occurs on this path. Figure 131 shows an automaton that covers all paths from $e1$ to $e2$ along invariant (1) on which no LdCnt(p1) events occur. All paths meet the right hand side of Property 1.

Property 2 *Let $e1$ be an event with an* a.p1 *condition $c1$ in its postset and $e2$ be an event with an* a.p1 *condition $c2$ in its preset, and let there be a path of program causality from $e1$ to $e2$ on which no* LdCnt *event*

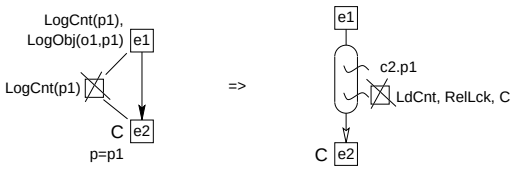


Figure 130: Property 1

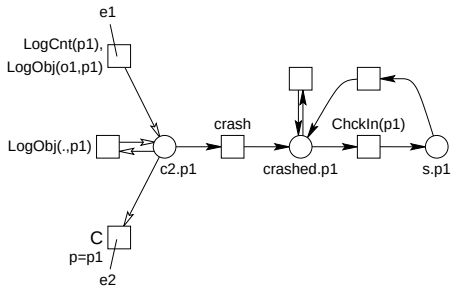


Figure 131: Property 1: Automaton for (1) paths from $e1$ to$e2$

occurs. Then, all conditions occurring on the path of program causality from $e1$ *to* $e2$ *are* a.p1 *conditions. Moreover, there is no* LdCnt(p1) *event between* $e1$ *and* $e2$.

Figure 132 shows a graphical representation of Property 2.

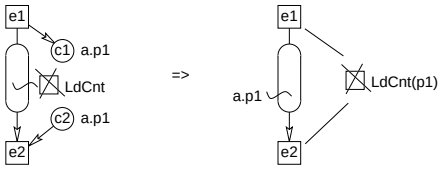


Figure 132: Property 2

The automaton from Fig. 133 shows all paths of program causality from $e1$ to $e2$ on which no LdCnt events occur. Note that there are three instances for $e1$ in this automaton: one instance with an arc of program causality from $e1$ to an a.p1 condition, one instance with an arc of program causality to a cnt.p1 condition, and one instance with an arc of program causality to a log.(.,cnt,p1) condition. But, only the first instance establishes a path of program causality to $e2$ (on which no LdCnt event occurs). Thus, on all paths of this automaton from $e1$ to $e2$, there are only a.p1 conditions. It remains to show that there is no LdCnt(p1) event between $e1$ and $e2$: A LdCnt(p1) event e has a s.p1 condition in its preset and a a.p1 condition in its postset. By invariant (1), this event cannot occur between $e1$ and $e2$—otherwise a s.p1 and a a.p1 are concurrent, which violates the invariant.

Property 3 *Let* $e1$ *be an event with an* a.p1 *condition* $c1$ *in its postset, and let* $e2$ *be an event with an* a.p1 *condition* $c2$ *in its preset. Furthermore, let* $e1$ *occur causally before* $e2$ *with no intermediate* LdCnt(p1) *event. Then, there is a path of program causality from* $c1$ *to* $c2$ *on which no* LdCnt *event occurs.*

Figure 134 shows a graphical representation of Property 3.

By invariant (1), we know that there is a path of causality from $e1$ to $e2$ on which only conditions of this invariant occur. The automaton from Fig. 135 shows all these paths on which no LdCnt(p1) events occur, which are excluded by assumption. All paths from $e1$ to $e2$ are paths of program causality on which only conditions a.p1 occur. This proves Property 3.

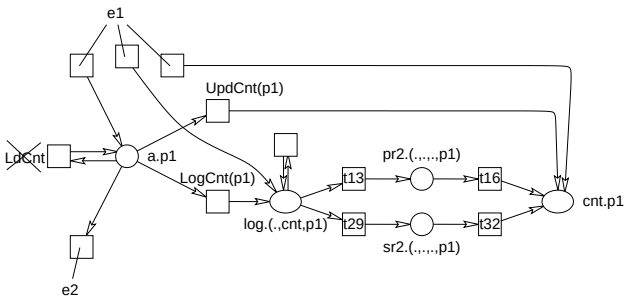


Figure 133: Property 2: Automaton for paths of program causality from $e1$ to $e2$

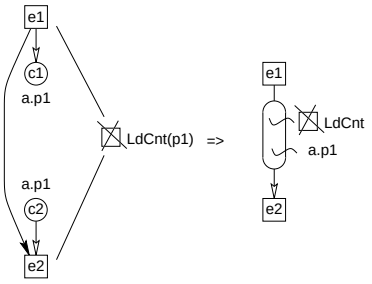


Figure 134: Property 3

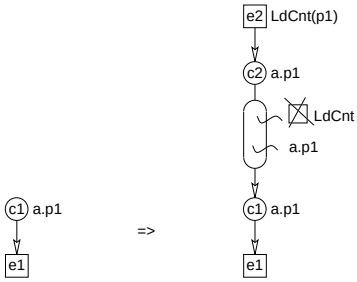


Figure 135: Automaton proving Property 3

Property 4 *Let $e1$ be an event with an* a.p1 *condition $c1$ in its preset. Then, there exists a* LdCnt(p1) *event $e2$ with a path of program causality from $e2$ to $c1$ on which only* a.p1 *conditions occur and on which no* LdCnt(p1) *event occurs.*

Figure 136 shows a graphical representation of Property 4.

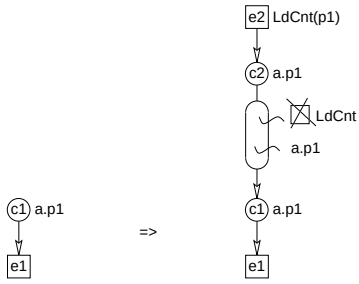


Figure 136: Property 4

The automaton from Fig. 137 shows the chain of causes of event $e1$ and immediately proves the property.

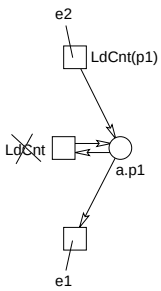


Figure 137: Property 4: Chain of causes of event $e1$

Property 5 *1. Let $e1$ be an event with a* lc *condition in its context. Then, event $e1$ has a* lc.j' *condition in its preset and a* lc.j *condition in its postset for some j and j' with $j' \leq j \leq j' + 1$.*

2. Let $e1$ and $e2$ be two events that both have an lc *condition in its context. Then, event $e1$ and $e2$ are causally ordered in one way or the other.*

3. Let $e1$ be an event with a lc.j' *condition in its preset and a* lc.j *condition in its postset, and let $e2$ be an event with a* lc.k *condition in its preset and a* lc.k' *condition in its postset. Furthermore, let $e1$ occur causally before $e2$ (see Fig. 138. Then we have $e1 = e2$ or we have $j \leq k$.*

Figure 138 shows the graphical representation of Property 5.3. Property 5.1 follows immediately from

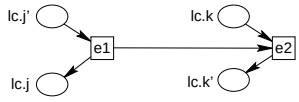


Figure 138: Property 5.3: $e1 = e2$ or $j \leq k$

the structure of the transitions of the Petri net model: Each transition which accesses the log counter lc in its preset and its postset. Moreover, each transition either increments the value or leaves it unchanged.

Property 5.2 follows immediately from invariant (10), which says that there is exactly one token on place lc. If the events $e1$ and $e2$ were not causally ordered, invariant (10) would be violated.

Property 5.3 follows from the automaton shown in Fig. 139: In case $e1 \neq e2$ we know that there is a path from the lc.j condition to the lc.k condition along invariant (10); on this path, the counter i is only increased. Therefore, we have $k \geq j$.

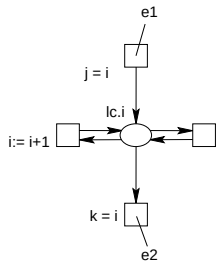


Figure 139: Property 5.3: Automaton along invariant (10)

Property 6 *1. Let c be a* log.(j+1,z,p1) *condition for some value z and some process $p1$. Then, there exists an event e that e has a* lc.j *condition in its preset and has a* lc.j+1 *condition in its postset such that there exists a path of causality from e to c on which only* lc.(j+1,z,p1) *conditions occur. Let us call this event e a log event e for the* log.(j+1,z,p1) *condition c. For $z = o1$ for some object $o1$, event e is a* LogObj(o1,p1) *event. For $z = cnt$, e is a* LogCnt(p1) *event. For $z = cmt$, e is a commit event with $p = p1$. For $z = dbu$, e is a* LogDBu(p1) *event, a* t17 *event with $p = p1$, or a* t33 *event with $p = p1$.*

2. Let $c1$ be a log.(j+1,z1,p1) *condition, and let $c2$ be a* log.(j+1,z2,p2) *condition. Moreover, let $e1$ be a log event for $c1$, and let $e2$ be a log event for $c2$ (see a.). Then, we have $e1 = e2$.*

In particular, we have $z1 = z2$ and $p1 = p2$. Moreover, for each log condition, the log event is unique.

Property 6.1 follows from the automaton shown in Fig. 140, which shows the chain of causes of a log.(j+1,z,p1) condition. The arguments for Property 6.2 are as follows: By Property 6.1, we know that

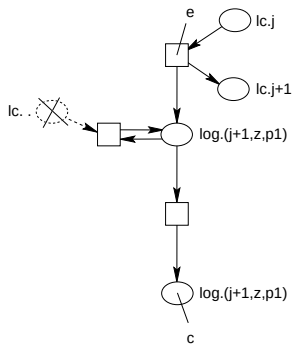


Figure 140: Property 6.1: A chain of causes for a log.(j+1,z,p1) condition

for $c1$ there is a log event $e1$ with a lc.j condition in its preset and a lc.j+1 condition in its postset. By the same argument, we have a log event $e2$ for $c2$ with a lc.j in its preset and a lc.j+1 in its postset. Let us assume $e1 \neq e2$. By Property BP:LogOrder.2, we know that $e1$ and $e2$ are causally ordered in one way or the other. In either case, Property 5.3 gives us $k + 1 \leq k$ — a contradiction. Thus, we have $e1 = e2$.

Property 7 *1. For* W[o1] *event $e1$ with $p = p1$ and each* LogObj(o1,p1) *$e1$, there exists a* WLck(o1,p1) *event $e2$ and a* LdCpy(o1,p1) *event $e3$ as shown in Fig. 141.*

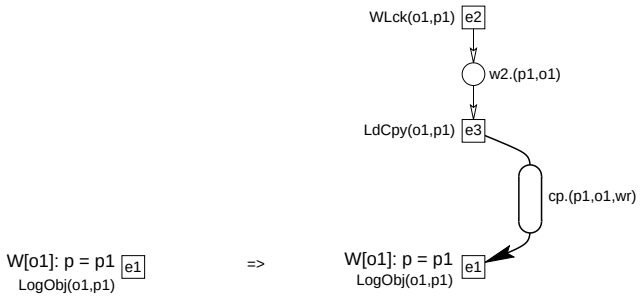


Figure 141: Property 7.1

2. *For each* R[o1] *event* $e1$ *with* $p = p1$*, there exists a* Lck(o1,p1) *event* $e2$ *and a* LdCpy(o1,p1) *event* $e3$ *as shown in Fig. 142.*

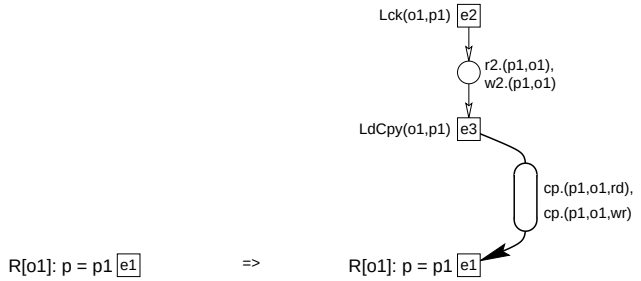


Figure 142: Property 7.2

3. *For each* UpdObj(o1,p1) *event* $e1$*, there exists a* LogObj(o1,p1) *event* $e2$ *as shown in Fig. 143.*

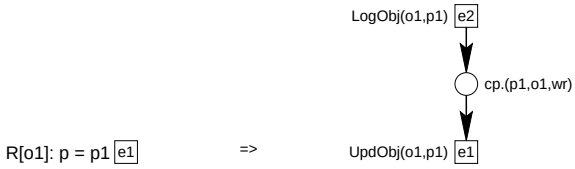
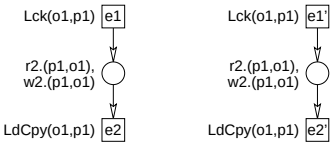


Figure 143: Property 7.3

Property 7.1 and 7.2 follow from the automata shown in Fig. 144, which represent a chain of causes of the corresponding events. Property 7.3 follows immediately from the Petri net model—therefore, we omit the corresponding automaton, which is almost identical to the situation in Fig. 143.

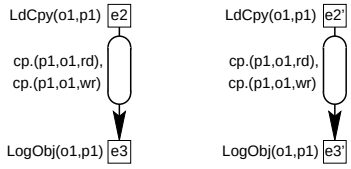
Property 8 1. *In the following situation*



we have $e1 = e1'$ *if and only if* $e2 = e2'$*.*

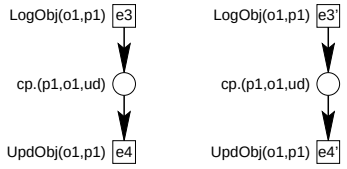
2. *In the following situation*

Figure 144: Automata for proving Property 7.1 and Prop 7.2



we have $e2 = e2'$ *if and only if* $e3 = e3'$.

3. *In the following situation*



we have $e3 = e3'$ *if and only if* $e4 = e4'$.

Here, we only proof Property 8.2; the other proof are analog: First, we assume $e2 = e2'$. We must show that $e3 = e3'$. Let us assume to the contrary that $e3 \neq e3'$. We know that $e3$ and $e3'$ occur causally after 2 and on the same path of causality along invariant (5). Without loss of generality, we assume that $e3$ occurs first. Thus, we have a path cp.(p1,o1,wr) conditions from $e2$ via $e3$ to $e3'$. This, however, is impossible since $e3$ has a cp.(p1,o1,wr) condition in its preset, but not in its postset. Second, we assume $e3 = e3'$. By the same arguments, we know there is a path of cp.(p1,o1,wr) conditions from $e2$ to $e2'$ (or vice versa). Again, this is impossible, because $e2$ and $e2'$ have a cp.(o1,p1,wr) condition in their post set but not in their preset.

The arguments for the other properties are the same; the only difference is that we use r2.(p1,o1), w2.(p1,o1) chains or cp.(p1,o1,ud) chains in the argument respectively.

Property 9

1. *If there is a path of* r2.(p1,o1), w2.(p1,o1), *and* cp.(p1,o1,.) *conditions from an event* $e1$ *to an event* $e2 \neq e1$, *then there is no* LdCnt(p1), *no* LogDBu(p1), *no* UnLck(o1,p1), *and no* RelLck(p1) *event between* $e1$ *and* $e2$ *(see Fig. 145).*

2. *Let there be two events* $e1$ *and* $e3$ *such that* $e1$ *occurs before* $e2$, *and such that* $e1$ *has a* a.p1 *condition and a* k.(p1,[..(o1,wr)..]) *condition in its postset, and such that* $e2$ *has a* a.p1 *condition and a* k.(p1,.) *condition in its preset. If there is no* LdCnt(p1) *event and no* RelLck(p1) *event between* $e1$ *and* $e3$, *then there is a path of* k.(p1,[..(o1,wr)..]) *conditions between* $e1$ *and* $e2$ *(see Fig. 146).*

3. *Let there be three events* $e1$, $e2$, *and* $e3$ *such that* $e1$ *occurs before* $e2$, *and such that* $e2$ *occurs before* $e3$. *Moreover, let there be a* a.p1 *condition and a* k.(p1,[..(o1,x)..]) *in the postset of* $e1$, *and let there be a* a.p1 *condition and a* k.(p1,.) *condition in the preset of* $e3$. *If there is no* LdCnt(p1) *event between* $e1$

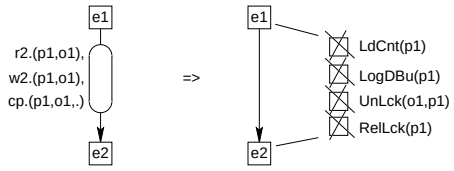


Figure 145: Property 9.1

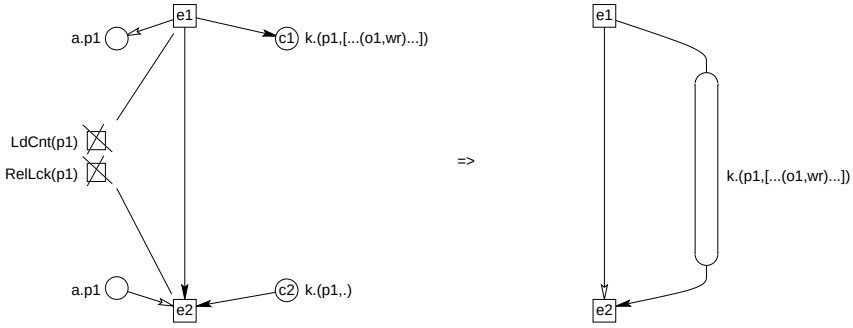


Figure 146: Property 9.2

and $e3$*, and if there are no* RelLck(p1) *and* UnLck(o1,p1) *events between* $e1$ *and* $e2$*, then there exists an event* $e4$ *causally between* $e2$ *and* $e3$ *such that there is a path of* k.(p1,[..(o1,x)..]) *conditions between* $e1$ *and* $e4$ *(see Fig. 147).*

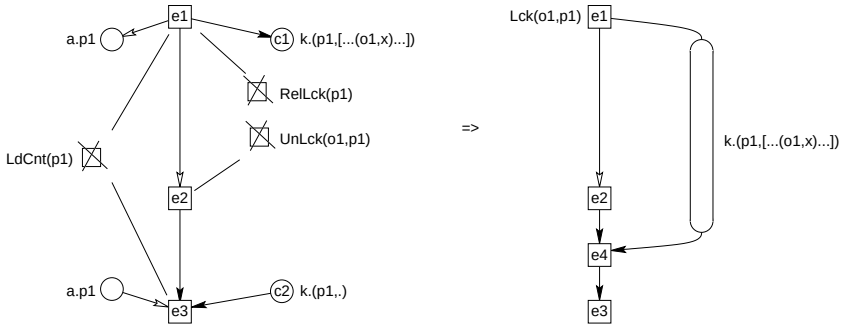


Figure 147: Property 9.3

Proof Property 9.1 : We know by invariant (5), that LdCnt(p1), LogDBu(p1) and UnLck(o1,p1) events cannot occur between $e1$ and $e2$—otherwise a s.p1 condition or a ul.(p1,o1) is concurrent to one of the w2.(p1,o1), r2.(p1,o1), or cp.(p1,o1,.) conditions on the path from $e1$ to $e2$. It remains to show that there is no RelLck(p1) event between $e1$ and $e2$: Each event that has a w2.(p1,o1), r2.(p1,o1), or cp.(p1,o1,.) condition in its postset has also a.p1 in its postset. Thus, $e1$ has a a.p1 condition in its postset (remember that $e1 \neq e2$). Let us consider a RelLck(p1) event e and a chain of its causes as shown in Fig. 148. We must show that this event e occurs before $e1$ or after $e2$: Remember that e' does not occur between $e1$ and $e2$ as shown above; i.e. e' occurs before $e1$ or occurs after $e2$. In the first case, e occurs also after $e2$, which finishes the proof. Let us consider the second case: First, we know that $e1$ and e' are different. Moreover, there is a unique path of causality from e' to $e1$ along invariant (1). The next event on this path from e' to $e1$ is the RelLck(p1) event e (see Fig. 148). By $e' \neq e1$, we know that e occurs before $e1$ on this path.

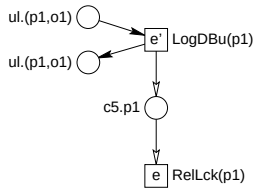


Figure 148: Property 7.1: Causes of a RelLck(p1) event

Proof Property 9.2 By invariant (2), we know that there is a unique path of causality from $c1$ via $c2$ to $e3$ along this path. The automaton from Fig. 149 shows all paths from $c1$ to $e3$ along invariant (2) on which no UnLck(o1,p1) and RelLck(o1) events occur—remember that these are excluded by assumption. This automaton shows that we either have a path of k(p1,[..(o1,wr)..]) conditions from $e1$ to $e2$, or we have a CheckIn(p1) event between $e1$ and $e2$. This, however, is impossible: Remember that a CheckIn(p1) event has a cr.p1 condition in its preset. On the other hand, we know, by Property 3, that there is a path of a.p1 conditions from $e1$ to $e2$. Thus, an occurrence of CheckIn(p1) between $e1$ and $e2$ would violate invariant (1).

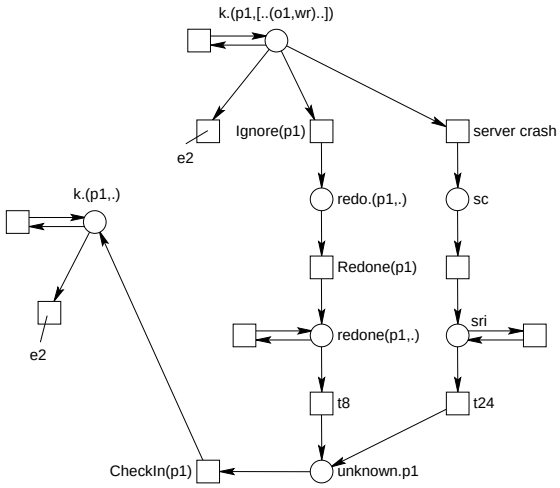


Figure 149: Property 9.2: Automaton for paths from $c1$ to $e2$ along invariant (2)

Proof Property 9.3 By invariant (2), we know that there is a unique path of causality from $c1$ via $c2$ to $e3$ along this path. The automaton from Fig. 149 shows all paths from $c1$ to $e3$ along invariant (2). On the paths with the UnLck(o1,p1) or the RelLck(p1) event $e4$, the proof is finished because we know that $e4$ does not occur between $e1$ and $e2$ by assumption; thus $e4$ occurs after $e2$. So it remains to consider the other cases: In these other cases, we know that we have a CheckIn(p1) event between $e1$ and $e3$. We will show that this impossible. Remember that a CheckIn(p1) event has a cr.p1 condition in its preset. On the other hand, we know, by Property 3, that there is a path of a.p1 conditions from $e1$ to $e3$. Thus, an occurrence of CheckIn(p1) between $e1$ and $e3$ would violate invariant (1).

Property 10 *The implication shown in Fig. 151 holds.*

Proof Property 10: Let us consider the situation on the right hand side of Fig. 151. By Property 7.1 we immediately get the situation shown in Fig. 152. Moreover, we know that there are no LdCnt(p1) events and RelLck(p1) events between $e4$ and $e1$ by Property 148. By Property 2, we know that there is a path

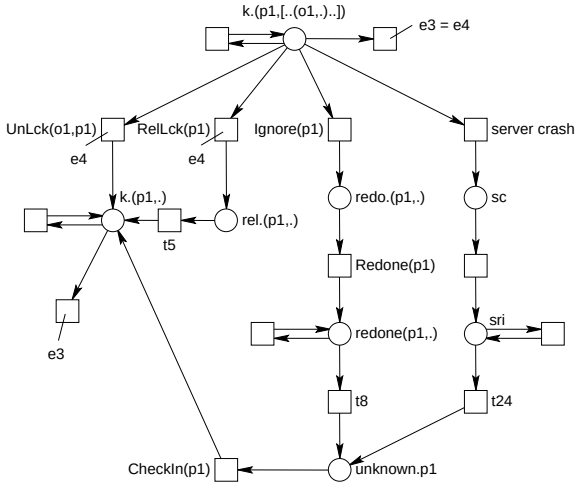


Figure 150: Property 9.3: Automaton for paths from $c1$ to $e3$ along invariant (2)

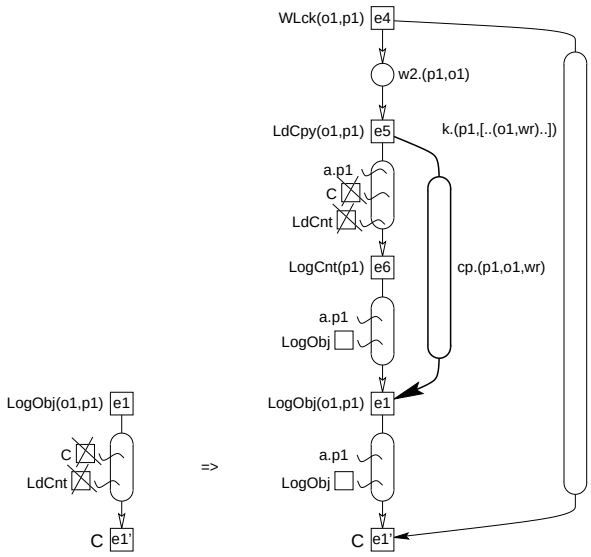


Figure 151: Property 10: Context of a directly committed LogObj(o1,p1) event

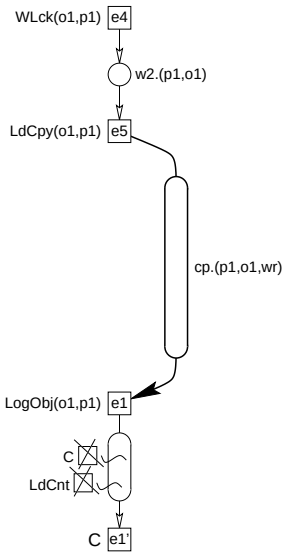


Figure 152: Situation after applying Property 7.1

of a.p1 conditions between $e5$ to $e1$ The automaton from Fig. 153 shows all possible paths from $e5$ to $e1$ along a.p1; moreover it shows all possible paths from $e1$ to $e1'$. This gives us the situation from Fig. 154,

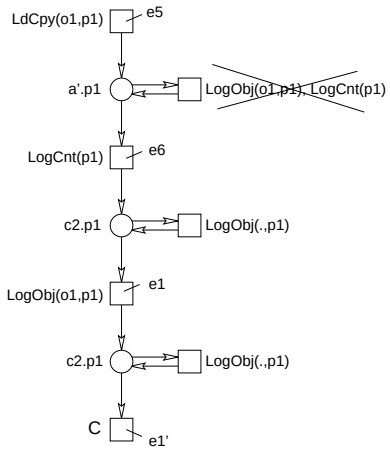


Figure 153: Automaton for paths from $e5$ to $e1'$ along a.p1 conditions

where we have added the k.(p1,.) condition in the context of $e4$ and $e1'$ that follows immediately from the Petri model. Since there are only LogObj events on the path from $e1$ to $e1$, we know that there are no RelLck(p1) and LdCnt(p1) events between $e1$ and $e1'$—otherwise invariant (1) is violated. Altogether, there are no RelLck(p1) and LdCnt(p1) events between $e4$ and $e1'$. By Property 9.2, we know that there is a path of k.(p1,[..(o1,wr)..]) conditions between $e4$ and $e1$, which gives us the situation on the right hand side of Fig. 151.

8.4 Assumptions on the executions

In our specification and in the proof, we made some assumptions on the executions. Some of these assumptions are explicit in the definition of an execution of a Petri net model (e. g. the acyclicity of causalities).

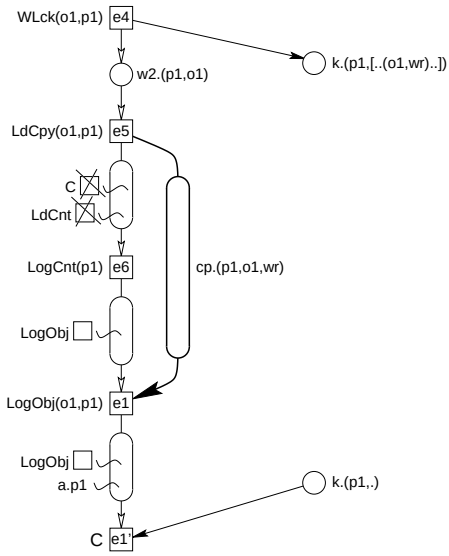


Figure 154: Situation after applying the path automaton

Other assumptions, however, must still be checked for our Petri net model. Here, we verify these remaining assumptions.

We start with the assumption that a path of data causality always concerns the same object, and that a path of program causality always concerns the same process. For short, we say that a path of causality is on the same instance of an object or of a process, respectively.

Property 11 (Causalities are on the same instance) *1. Let $e1$ be an access event on some object $o1$, and let $e2$ be an access event on some object $o2$. If there is a path of data causality from $e1$ to $e2$, then we have $o1 = o2$.*

2. Let $e1$ be a process event of process $p1$, and let $e2$ be a program event of process $p2$. If there is a path of program causality from $e1$ to $e2$, then we have $p1 = p2$.

This assumption is graphically represented in Fig. 155. This property can be easily checked, by following

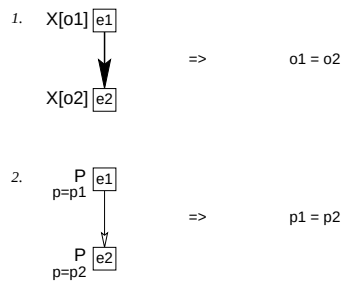


Figure 155: Property 11: Causality are on the same instance

the paths of data causality or program causality in the Petri net model. On these paths, the value of o resp. p does never. Formally, this could be proven by the corresponding automata: For data causality, we can use the automaton from Fig. 88 on page 76. For program causality, we can use the automaton from Fig. 60 on page 56.

The second assumption is that each path of data causality originates from the corresponding object in the initial marking of place objects—its initial value in the database. Likewise, we assumed that each path

of program causality originates in the initial marking of place continuations—the initial continuation of the process. We formalize this property local to the context of each condition and each event.

Property 12 (Initialized paths) *Let $e1$ and $e2$ be two events, let c be a condition such that $e1$ is in the preset of c and $e2$ is in the postset of c.*

1. *Let $e1$ and $e2$ be two events, let c be a condition such that $e1$ is in the preset of c and $e2$ is in the postset of c. If the arc from c to $e2$ is a data causality, then the arc from $e1$ to c is also a data causality.*
2. *Let $e1$ and $e2$ be two events, let c be a condition such that $e1$ is in the preset of c and $e2$ is in the postset of c. If the arc from c to $e2$ is a program causality, then the arc from $e1$ to c is also a program causality.*
3. *Let e be an event with an arc of data causality to some condition $c1$, then there exists a condition $c2$ in the preset of $e1$ with an arc of data causality from $c2$ to e.*
4. *Let e be an event with an arc of program causality to some condition $c1$, then there exists a condition $c2$ in the preset of $e1$ with an arc of program causality from $c2$ to e.*

This assumption is graphically represented in Fig. 156. The proof of this assumption is straight-forward.

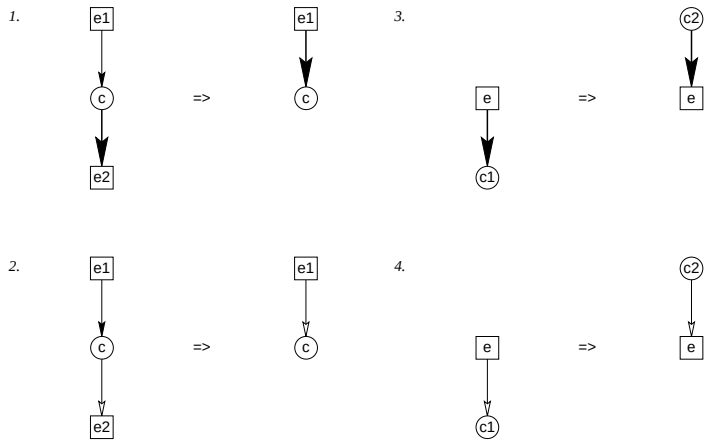


Figure 156: Property 12: Initialized paths

The third and forth item immediately follows from the structure of the transitions of the Petri net model: Each transition with an outgoing data causality arc has in incoming data causality arc; the same holds for program causality arcs.

The proof of the first item is similar: Except for place objects each place with an outgoing data causality arc has only incoming data causality arcs. So, the only condition c that could violate the property is a objects.(o1,v) condition. Let us assume we have such a condition c in an execution that violates item 1: Then, there is an incoming arc of this condition c that is no data causality. Furthermore, there is an outgoing arc of this condition that is a data causality. From the Petri net model, we know that the only in-going arc of c that is no data causality comes from an arc label by $(o1, invalid)$—this implies $v = invalid$. On the other hand, each out-going data causality arc is labeled by $(o1, valid)$. Thus, we have $v = valid$. Which is a contradiction.

The arguments for item 2 are analog for place continuations.

9 Conclusion

We have presented a formal proof for a design pattern for the fault-tolerant execution of long-running parallel programs. The proof is admittedly long and comprises many cases, but its parts and the applied methods are simple: automata for distinguishing the relevant cases and for adding further details to a considered situation. This way, the implication between situations are verified in a constructive way.

Up to now, the proof is hand-made. But, finding the corresponding automata for a given Petri net is straight-forward in most cases. Given the automaton, checking the arguments is arduous but not difficult. Basically, it is the vast number of different cases that intimidate a human proof checker. This task, however, could be solved by automated theorem provers in a way similar to the proof checkers proposed in [3]—in fact, some tasks can already be solved by this tool.

The development of a tool supporting the methods used in this paper is an ongoing project. In the long run, a tool should be able to construct and to check the arguments given in Sect. 7—the most tedious part of this paper. The hand-made proof in this paper is one milestone on the way towards this goal.

Acknowledgment

We would like to thank Jörn Freiheit for his comments on a preliminary version of this report.

References

[1] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[3] Thomas Baar, Ekkart Kindler, and Hagen Völzer. Verifying intuition – ILF checks DAWN proofs. In S. Donatelli and J. Kleijn, editors, *Application and Theory of Petri Nets 1999, 20th International Conference*, volume 1639 of *LNCS*, pages 404–423. Springer, June 1999.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] Eike Best and César Fernández. *Nonsequential Processes*, volume 13 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.

[6] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Springer, 2000. in preparation.

[7] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *13th Annual International Symposium on Computer Architecture*, pages 434–442. IEEE, June 1986.

[8] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. Sa survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, CMU, June 1999.

[9] Kathi Fisler and Claude Girault. Modelling and model checking a distributed shared memory system. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998, 19th International Conference*, volume 1420 of *LNCS*, pages 84–103. Springer-Verlag, June 1998.

[10] Rob Gerth. Sequential consistency and the lazy caching algorithm. *Distributed Computing*, 12:57–59, 1999.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennesy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[12] Dominik Gomm and Ekkart Kindler. Causality based specification and correctness proof of a virtually shared memory scheme. SFB-Bericht 342/6/91 B, Technische Universität München, August 1991.

[13] Dominik Gomm and Ekkart Kindler. A weakly coherent virtually shared memory scheme: Formal specification and analysis. SFB-Bericht 342/5/91 B, Technische Universität München, August 1991.

[14] Dominik Gomm and Ekkart Kindler. Causality based proof of a distributed shared memory system. In A. Bode and M. Dal Cin, editors, *Parallel Computer Architectures: Theory, Hardware, Software, Applications*, volume 732 of *LNCS*, pages 131–149. Springer-Verlag, 1993.

[15] Susanne Graf. Characterization of sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 12:75–90, 1999.

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

[17] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *10th International Conference on Distributed Computing Systems*, pages 302–309. IEEE, May 1990.

[18] Kurt Jensen. *Coloured Petri Nets, Volume 1: Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[19] Kurt Jensen. *Coloured Petri Nets. Volume 2: Analysis Methods*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 1995.

[20] Karpjoo Jeong, Dennis Shasha, Srendranath Talla, and Peter Wyckoff. An approach to fault-tolerant parallel processing on intermittently idle heterogeneous workstations. In *The 27th International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 11–20. IEEE, June 1997.

[21] Ekkart Kindler. A classification of consistency models. Technical Report B99-14, Freie Universität Berlin, Institut für Informatik, October 1999.

[22] Ekkart Kindler, Andreas Listl, and Rolf Walter. A specification method for transaction models with data replication. Informatik-Berichte 56, Humboldt-Universität zu Berlin, March 1996.

[23] Ekkart Kindler and Rolf Walter. Arc-typed Petri nets. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996*, volume 1091 of *LNCS*, pages 289–306. Springer-Verlag, June 1996.

[24] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[25] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[26] David Lomet and Gerhard Weikum. Efficient transparent application recovery in client-server information systems. In L. M. Haas and A Tiwary, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 460–471, Seattle, Washington, USA, June 1998. ACM Press.

[27] (Editor) Micheal Merritt. Special issue: Verification of lazy caching. *Distributed Computing*, 12(2/3), March 1999.

[28] Nuno Neves, Miguel Castro, and Paulo Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the thirteenth ACM Symposium on Principles of Distributed Systems, PODC '94*, pages 121–129. ACM, August 1993.

[29] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.

[30] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, May 1991.

[31] Fred. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[32] Andrew Xu and Barbara Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *International Symposium on Fault-Tolerant Computing (FTCS'89)*, volume 1420, pages 199–207. IEEE Computer Society Press, June 1989.