Ogre and Pythia: An Invariance Proof Method for Weak Consistency Models

Jade Alglave
Microsoft Research Cambridge, University College London, UK
jaalglav@microsoft.com, j_alglave@ucl.ac.uk

Abstract
We design an invariance proof method for concurrent programs parameterised by a weak consistency model. The calculational design of the invariance proof method is by abstract interpretation of a truly parallel analytic semantics. This generalises the methods by Lamport and Owicki-Gries for sequential consistency. We use a for an example of language to write consistency specifications of both concurrent programs and machine architectures. We use a as et al. to example of language to write consistency specifications of both concurrent programs and machine architectures. We use a as example of language to write consistency specifications of both concurrent programs and machine architectures. We use a as example of language to write consistency specifications of both concurrent programs and machine architectures.

Categories and Subject Descriptors D.1.3 (Concurrent Programming); D.2.4 (Verification); F.3.1 (Invariants); F.3.2 (Semantics).

Keywords concurrency, distributed and parallel programming, invariance, verification, weak consistency models.

When an ogre (Owicky-Gries' Extended) meets a pythia (variable) classic tales get retold: in this paper we investigate an invariance proof method for concurrent (parallel or distributed) algorithms parameterised by weak consistency models.

Different program semantics styles can be used to describe concurrent program executions, for example operational, denotational or axiomatic semantics. We introduce here a new style, that we call analytic: it is more abstract than operational models (Boudol et al. 2012) or pomsets (Brookes 1975). In this context, we separate the individual traces of the different processes that constitute the program from the communications between processes.

Weak consistency models (WCMs) are seen as placing more or less restrictions on communications. WCMs are now a fixture of computing systems: for example Intel x86 or ARM processors, Nvidia graphics cards, programming languages such as C++ or OpenCL, or distributed systems such as Amazon Web Services or Microsoft’s Azure. In this context, the execution of a concurrent program can be seen as an interleaving of the individual traces of the different processes that constitute the program, but the communication between processes are unlike what is prescribed by Lamport’s Sequential Consistency (SC) (Lamport 1979). Indeed the read of a shared variable may read another value than the one written by the last process writes (for example due to hardware features such as store buffers and caches).

Different consistency semantics styles can be used to describe WCMs. Operational models define abstract machines in which executions of programs are sequences of transitions made to or from formal entities modelling e.g. hardware features such as store buffers and caches. Axiomatic models abstract away from such concrete features and describe executions as relations over events modelling e.g. read or write memory accesses, and synchronisation.

We calculate an invariance proof method as an abstraction of the analytic semantics. Thus our method is parameterised by a WCM.

1. Overview of the Analytic Semantics
Our analytic semantics describes program executions as their anarchic semantics (process computations without any restriction on communications), and their communication semantics (restrictions on communication between processes). To illustrate our analytic semantics, we will use the load buffering example 1b in Fig. 1.

Figure 1: 1b algorithm in LISA

In 1b, processes P0 and P1 communicate via shared variables x and y (initialised to 0 at line 0). Each process reads a variable (x at line 1 for P0, and y at line 11 for P1), then writes to the other variable (y at line 2 for P0 and x at 12 for P1).

1.1 Anarchic Semantics
Fig. 2 gives one of the four anarchic executions of 1b. After initialising x and y to 0, the computations of P0 and P1 are formalised by traces, viz., finite or infinite sequences of states separated by unique events. States and events appear along a trace in the process execution order.

Events give a semantics to instructions, for example accesses to registers or memory locations. States record a process program point, the value of local variables (registers r1 for P0 and r2 for P1) and the value of pythia variables.

A pythia variable is the unique name given to the value of a read event, e.g. x1 for the read r2 at line 2 for P1. Our pythia variables are different from ghost variables; ghost variables compensate for objects that do not exist in the chosen program semantics.

The read-from relation rf describes communications between processes. In Fig. 2, the read r2 takes its value from the write w1 (so the value 1 is assigned to the pythia variable r2).

The interleaving of the processes’ executions is given by cuts. The sequence of cuts π0; π1; π2; π3; π4 in Fig. 2 formalises the interleaving x=0; y=0; r1 x; r1 y; r2 y; w1 x; w1 y. Contrary to operational models, the anarchic semantics does not define the coherence order i.e., the order in which the writes to a given memory location hit the shared memory, since this is part of the WCM. Independently of the order of execution of the write

Figure 2: One anarchic execution for 1b

\[ \begin{align*}
1: & \quad x = 0, y = 0; \\
2: & \quad r1 x \rightarrow x1; r1 y \rightarrow y1; \\
3: & \quad r2 y \rightarrow y2; \\
4: & \quad w1 x \rightarrow x2; w1 y \rightarrow y2; \\
5: & \quad r2 y \rightarrow y2; \\
\end{align*} \]
actions (defined by the cuts), all possible coherence orders are a priori possible (and will be considered in cat with with co from AllGo (Alglave et al. 2016)).

1.2 Communication Semantics

The communication semantics filters anarchic executions according to certain restrictions on the communication between processes (i.e., the read-from relation rf).

To apply these restrictions more easily, we abstract anarchic executions into candidate executions, where communicated values and cuts are abstracted away. A candidate execution consists of the set of events (partitioned into reads, writes—including the initialisation writes IW, tests, fences), the process execution order po (a total per process, between consecutive events on a trace), and the read-from relation rf. Fig. 3 shows the candidate execution which abstracts the anarchic execution of 1b of Fig. 2.

We use the domain-specific language cat (Alglave et al. 2016) as an example of a language to specify restrictions on communications. In cat, we can forbid the anarchic execution of 1b in Fig. 3 by asking its candidate execution abstraction in Fig. 3 to satisfy the constraint irreflexive po; rf po; rf. Thus the candidate execution of Fig. 3 should not have a reflexive sequence that alternates process execution order (po) and communications (rf). This is not the case since: $r_1^1 po w_3^1 rf y_1^1 po w_2^1 rf r_1^1$.

1.3 Invariance Semantics

We follow (Cousot and Cousot 1980) and define the invariance semantics by abstraction of the analytic semantics. The invariance semantics relates each local program point to the values of the other program points, local variables, pythia variables, and rf along all cuts of all executions going through that local program point. For example $S_{nom} \Rightarrow S_{nom}$ is invariant for 1b where $S_{nom} = (at\{3\} \wedge at\{13\}) \Rightarrow \neg(r_1 = 1 \land r_2 = 1)$ and the communication hypothesis $S_{com} = \{\langle w_1^2, r_1^1 \rangle, \langle w_2^1, r_1^1 \rangle\} \not\in rf$ excludes the case of Fig. 2 and 3. The verification conditions are formally derived by calculational design from the formal definition of the analytic semantics and proceed by induction along cuts. In addition to the initialisation, sequential, and non-interference proof, the main difference with (Owicki and Gries 1976; Lamport 1977) is the use of pythia variables and the read-from relation rf in assertions and the communication proof showing that rf is well-formed. This proof method design methodology is independent of the considered language. We apply it to the Litus Instruction Set Architecture (LISA) language (Alglave and Cousot 2016) of the herd7 tool (Alglave and Maranet 2015).

2. Overview of the Invariance Proof Method

We aim at developing correct algorithms for a wide variety of weak consistency models $M_0, \ldots, M_n$. Given an algorithm $A$ and a consistency model $M \in \{M_0, \ldots, M_n\}$, our method is articulated as follows—we detail each of these points in turn below, and show a graphical representation in Fig. 4:

1. Design the algorithm $A$, state its invariant specification $S_{inv}$ (see Sect. 2.1), and its communication specification $S_{com}$ (see Sect. 2.2).

2. We write $A$ in LISA, using LISA’s special fence synchronisation markers if needed, which allow to define in cat which

\begin{verbatim}
0: { w F1 false; w F2 false; w T 0; }
P0: 1:w[] F1 true
   2:w[] T 2
   3:do {i}
   4:  r[] R1 F2 {~ F2^i}
   5:  r[] R2 T {~ T^i}
   6:while R1 & R2 1 {iand}
   7: (* CS1 *)
   8: w[] F1 false
   9:

P1: 10:w[] F2 true
   11:w[] T 1;
   12:do {j}
   13:  r[] R3 F1; {~ F1^j}
   14:  r[] R4 T; {~ T^j}
   15:while R3 & R4 2 {jand}
   16: (* CS2 *)
   17:w[] F2 false;
   18:

Figure 5: Peterson algorithm in LISA
\end{verbatim}

program points (perhaps sets of program points) synchronisation is needed for correctness;

2. Prove the correctness $S_{inv} \Rightarrow S_{inv}$ of the algorithm $A$ w.r.t. the invariant specification $S_{inv}$, under the communication specification $S_{com}$ (see Sect. 2.3.1);

3. Prove that the consistency model $M$ guarantees the communication specification $S_{com}$ that we postulated for the correctness of algorithm $A$ (i.e., $M \Rightarrow S_{com}$, see Sect. 2.3.3 and Sect. 2.3.4).

To illustrate our preamble, we use the classical mutual exclusion algorithm of Peterson (Peterson 1981), which requires explicit synchronisation to be correct on WCMs.

2.1 Algorithm: Design and Specifications

2.1.1 Writing our running example

We give the code of Peterson’s algorithm in LISA in Fig. 5. The algorithm uses two shared flags, F1 for the first process P0 (resp. F2 for the second process P1), indicating that the process P0 (resp. P1) wants to enter its critical section. The shared turn T grants priority to the other process: when T is set to 1 (resp. 2), the priority is given to P0 (resp. P1).

Let’s look at the process P0: P0 busy-waits before entering its critical section (see the do instruction at line 3) until (see the while clause at line 6) the process P1 does not want to enter its critical section (viz., when F2=false, which in turn means R1=false thanks to the read at line 4) or if P1 has given priority to P0 by setting turn T to 1, which in turn means that R2=1 thanks to the read at line 5.

Sect. 2.4 details the syntax and semantics of the LISA language.

Annotations We placed a few annotations in our LISA code, to ensure the unicity of events in invariants and proofs:

- iteration counters: each loop is decorated with an iteration counter, e.g. i at line 3 for the first process and j at line 12: for the second process. The names (i and at line 6 and j and at 15) represent the iteration counter when exiting the loop.
- pythia variables: each read, at lines 4 and 5 for the first process, and lines 13 and 14 for the second process, is decorated with a pythia variable. A read r[] R x at line $\ell$ in the program, reading the variable x and placing its result into register R, is
decorated with the pythia variable \( \{ \sim x_0^n \} \), where \( n \) is the iteration counter (for nested loops we record all iteration counters of all surrounding loops).

### 2.2 Communication Specification \( S_{\text{com}} \)

The next step in our specification process consists in stating an invariant communication specification \( S_{\text{com}} \), expressing which interprocess communications are allowed for the algorithm \( A \).

### 2.2.1 Peterson can go wrong under WCMs

Under certain WCMs, such as x86-TSO or any weaker model, Peterson’s algorithm does not satisfy the mutual exclusion specification \( S_{\text{inv}} \) of Fig. 6.

To see this, consider Fig. 7a. The plain red arrows \( r \) are an informal representation of a communication scenario where:

- on process \( P_0 \), the read at line 4 reads the value that \( F_2 \) was initialised to, at line 0, so that \( R_1 \) contains \( \text{false} \). And, the read at line 5 reads from any write of \( T \), so that \( R_2 \) contains one of the values \( 0 \), \( 1 \), or \( 2 \), indifferently.
- on process \( P_1 \), the read at line 13 reads from the initial value of \( F_1 \) so that \( R_3 \) contains \( \text{false} \). The read at line 14 reads from 0, \( 11 \), or \( 2 \) so that \( R_4 \) contains 0, 1, or 2, indifferently.

In this situation (which is impossible under SC), both loop exit conditions can be true so that both processes can be simultaneously in their critical section, thus invalidating the specification \( S_{\text{inv}} \). Another erroneous behaviour is illustrated in Fig. 7b. The value of \( \ell \) and \( r \) follows:

- \( \ell = \text{false} \) \( \land \) \( r = \text{true} \) \( \land \) \( i = j \), i.e., the communication \( \ell \) \( \land \) \( r \) \( \land \) \( i = j \) is indeed stronger than \( \ell \) \( \land \) \( r \) \( \land \) \( i \neq j \) because

\[\begin{align*}
&\ell = \text{false} \land r = \text{true} \land i = j \\
&\Rightarrow \ell = \text{false} \land r = \text{true} \land i \neq j,
\end{align*}\]

**Figure 6:** Invariant specification \( S_{\text{inv}} \) for Peterson’s algorithm

**Figure 7:** Incorrect executions of Peterson algorithm with WCM

F2 and F1 is indiffent. But process P0 reads T in R2 from the write 11 so R2=1 while P1 reads T in R4 from the write 2 so R4=2. In this situation (impossible under SC) the turns are wrong, so that both processes can be simultaneously in their critical section, again invalidating the specification \( S_{\text{inv}} \).

### 2.2.2 Communication specification \( S_{\text{com}} \)

Let us express the communication scenarios depicted in Fig. 7 as an invariant. We write the pythia triple \( r(f(x), \ell, v) \to m \) to mean that the read event \( r = \ell' : r([\ell] R x \sim x_0) \), or more precisely its unique pythia variable \( x_0 \) takes its value \( v \) from evaluating the expression \( e \) of the write event \( w = \ell' : \{ w([\ell] x \sim v) \} \to r \delta \ell \) and \( \ell' \) and \( x_0 \) and \( x_0 = v \) at \( \ell' \). The communication verification conditions will check that \( w, r \in r \) and the local invariant at \( \ell \) implies that \( e = v \). We define our communication specification \( S_{\text{com}} \) as follows:

\[ S_{\text{com}} \triangleq \neg \exists i, j. r(F2^i \land 0, \text{false}) \lor r(F2^j, 17, \text{false}) \lor r(F3^1, 11, 1) \land r(F1^1, 0, \text{false}) \land \neg r(F1^1, 8, \text{false}) \lor r(F1^1, 2, 2)]\]

\( S_{\text{com}} \) states the read-froms should yield values in the registers ensuring that both processes may not simultaneously leave their waiting loops. The scenarios in Fig. 7 are therefore impossible. This ensures that both processes cannot be simultaneously in their critical section.

Therefore, there cannot be two iteration counters \( i \) and \( j \) such that:

- The first process \( P_0 \) enters its critical section at the \( j \)th iteration of its waiting loop (corresponding to the pythia variables \( F2^i \) and \( T_j \)) because

  - either the read at line 4 and \( i \)th iteration (corresponding to the pythia variable \( F2^i \)) takes its value, \( \text{false} \), from the initialisation of the variable \( F2 \) (in the prelude at line 0) or from the write to \( F2 \) at line 17;
  - or, the read at line 5 and \( i \)th iteration (corresponding to the pythia variable \( T_j \)) takes its value, 1, from the write at line 11;

- And the second process \( P_1 \) enters its critical section at the \( j \)th iteration of its waiting loop (corresponding to the pythia variables \( F1^i \) and \( T_{j+1} \)) because

  - either the read at line 13 and \( j \)th iteration (corresponding to the pythia variable \( F1^i \)) takes its value, \( \text{false} \), from the initialisation of the variable \( F1 \) (in the prelude at line 0) or from the write to \( F1 \) at line 8;
  - or, the read at line 14 and \( j \)th iteration (corresponding to the pythia variable \( T_{j+1} \)) takes its value, 2, from the write at line 2.

\( S_{\text{com}} \) expresses hypotheses on the communications made by the threads of the program. \( S_{\text{com}} \) is independent from any consistency models. \( S_{\text{com}} \) is the weakest communication invariant since weakening any of its hypotheses provides a counter-example. \( S_{\text{com}} \) belongs to the abstract domain of invariants.

### 2.3 Our Proof Method

Recall Fig. 4; given an algorithm \( A \), an invariant specification \( S_{\text{inv}} \), a communication specification \( S_{\text{com}} \), and a WCM \( M \) we have to prove \( M \Rightarrow S_{\text{inv}} \). Our method is articulated as follows:

1. **Conditional invariance proof** \( S_{\text{com}} \Rightarrow S_{\text{inv}} \): we prove that if the communications occur like prescribed by \( S_{\text{com}} \), then the processes satisfy the invariant \( S_{\text{inv}} \);
2. **Inclusion proof** \( M \Rightarrow S_{\text{com}} \): we prove that the WCM \( M \) guarantees the communication hypotheses made in \( S_{\text{com}} \).

We now detail each proof in turn.

#### 2.3.1 Conditional invariance proof \( S_{\text{com}} \Rightarrow S_{\text{inv}} \)

We have to prove that each process of the algorithm \( A \) satisfies the invariant \( S_{\text{inv}} \) under the hypothesis \( S_{\text{com}} \); to do so we:

1. invent a stronger invariant \( S_{\text{red}} \), which is inductive;
2. prove that \( S_{\text{red}} \) is indeed inductive, i.e., satisfies verification conditions implying that if it is true, it stays true after one step of computation or a communication that satisfies \( S_{\text{com}} \); effectively we prove \( S_{\text{com}} \Rightarrow S_{\text{red}} \);
3. prove that \( S_{\text{red}} \) is indeed stronger than \( S_{\text{inv}} \) (i.e., \( S_{\text{red}} \Rightarrow S_{\text{inv}} \)). From \( S_{\text{com}} \Rightarrow S_{\text{red}} \) and \( S_{\text{red}} \Rightarrow S_{\text{inv}} \) we conclude that \( S_{\text{com}} \Rightarrow S_{\text{inv}} \), which was our goal. We now illustrate the correctness proof method on Peterson.

- **An inductive invariant** \( S_{\text{red}} \) **stronger than** \( S_{\text{inv}} \) is given in Fig. 8 as local invariants (depicted in blue in curly brackets) for each program point of each process. Each local invariant attached to a program point can depend on the program state that is on registers (both the ones local to the process under scrutiny, and from other processes), pythia variables and, as in (Lamport 1977), on the program counter of the other processes. In general the local invariants may also depend on the possible communications \( r,i,e \), which reads may read their values from which writes (but this is not necessary in Fig. 8 since the program logic does not restricts in any way the possible communications as, e.g., would be the case for unreachable reads or writes).
that some value does not make any distinction on these cases and just states the invariant or, any value carried by the pythia variables. More precisely, the read-

\[ \text{read} \]

for \( \text{fi} \)

rently, and \( \text{tially, a} \)

that the invariants hold when executing one process sequentially, a proof

\[ \text{proof} \]

Sequential hypothesis is made on communications and therefore no possible communication allowed by the communication invariant \( \text{com} \). We prove \( \text{com} \) from the communication specification \( \text{com} \). This is an abstraction since e.g. in \( \text{cat} \) shared variable names and their values are abstracted away. So, in general, \( \text{com} \) will allow less behaviors that \( \text{com} \) and the \( \text{M}_i \) less that \( \text{com} \). The proof \( \text{com} \) proceeds as follows:

- we build the communication scenarios corresponding to the pythia triples given in the communication specification \( \text{com} \) from an anarchic invariant \( \text{com} \);
- we write a consistency specification \( \text{com} \) (e.g. in \( \text{cat} \)) which will forbid each of these communication scenarios.

We illustrate the proof method with Peterson’s algorithm of Fig. 5 using \( \text{com} \) in (1) of Sect. 2.2.2. \( \text{com} \) in Fig. 8, and the consistency specification language \( \text{cat} \) which requires reasoning on candidate executions (see Sect. 11).

\[ \text{building the communication scenarios corresponding to the pythia triples for} \text{cat} \text{requires us building several candidate executions involving relations between accesses (i.e., read/write events) as follows (we illustrate on case 1 of Fig. 10).} \]

- read-from \( \text{rf} \); for each pythia triple, we depict the read-from relation \( \text{rf} \) in red; for example for \( \text{rf} \{ \text{F2}_1, \ 0, \ false \} \), we create a read-from relation between the initial write of \( \text{false} \) to the variable \( \text{F2} \) at line 0 and the read of \( \text{F2} \) from line 4, at the \( i \)th iteration.
- program order \( \text{po} \); we also depict the program order edges between the accesses which are either the source or the target of a communication edge (viz., read-from and coherence). In case 1 of Fig. 10, the po edges in purple are between the lines 1 and 4 on process P0, and lines 10 and 13 on process P0. po is irreflexive and transitive (not represented on Fig. 10).

\[
\begin{align*}
\text{at} \ 7 & \land \ \text{at} \ 16 \land \text{com} \\
& \Rightarrow (-\text{F2}_4\text{end} \lor T_{14}^\text{end} = 1) \land (-\text{F1}_3\text{end} \lor T_{14}^\text{end} = 16) \]
\end{align*}
\]

\( (i.e., \text{the invariant} \text{com} \text{holds at lines} 7 \text{and} 16) \) \[ \Rightarrow -\text{com} \]

\( \text{since by taking} \ i = \text{end} \text{and} \ j = \text{end}, \ \text{we have} \]

\( (\text{F2}_4\text{end} \lor T_{14}^\text{end} = 1) \land (\text{F1}_3\text{end} \lor T_{14}^\text{end} = 12) \)

Note that this calculation of \( \text{com} \) from the specification \( \text{com} \) and the anarchic inductive invariant \( \text{com} \) provides a formal method to discover \( \text{com} \) by calculational design. \( \text{com} \) is sufficient but also necessary, hence the weakest communication hypothesis, since for each possible case of communication excluded by \( \text{com} \), it is possible to find a counter-example execution of Peterson violating mutual exclusion (see Fig. 7 and 10).

2.3.2 WCM specification \( \text{com} \)

We have proved \( \text{com} \Rightarrow \text{com} \). To ensure that \( \text{com} \) holds in the context of the consistency model \( M \), we must prove \( M \Rightarrow \text{com} \), i.e., that all the behaviours allowed by \( M \) are allowed by \( \text{com} \). In general we have to consider several WCMs \( M = \{ M_i \}, i \in [0, n] \). To factorize the proofs \( \forall i \in [1, n] \cdot M_i \Rightarrow \text{com} \), we look for a (preferably weakest else minimal) consistency specification \( \text{com} \) that encompasses our specification \( \text{com} \). We prove \( \text{com} \Rightarrow \text{com} \) and then \( \forall i \in [1, n] \cdot M_i \Rightarrow \text{com} \), which are the only bits of proof that must be adapted when considering different models.

2.3.3 Inclusion proof \( \text{com} \Rightarrow \text{com} \)

As illustrated in Fig. 9, the WCMs \( \text{com} \) and \( M_i \), \( i \in [1, n] \) belong to the domain of consistency specifications (e.g. candidate executions for \( \text{cat} \)) while \( \text{com} \) belongs to the different domain of invariants. The proof \( \text{com} \Rightarrow \text{com} \) must therefore be done in the most abstract domain more concrete than both of these domains, which is the semantic domain of sets of executions. The same way that we derived \( \text{com} \) from the program specification \( -\text{com} \), we derive \( \text{com} \) from the communication specification \( -\text{com} \). This is an abstraction since e.g. in \( \text{cat} \) shared variable names and their values are abstracted away. So, in general, \( \text{com} \) will allow less behaviors that \( \text{com} \) and the \( M_i \) less that \( \text{com} \). The proof \( \text{com} \Rightarrow \text{com} \) proceeds as follows:

- we build the communication scenarios corresponding to the pythia triples given in the communication specification \( -\text{com} \) from an anarchic invariant \( \text{com} \);
- we write a consistency specification \( \text{com} \) (e.g. in \( \text{cat} \)) which will forbid each of these communication scenarios.

We illustrate the proof method with Peterson’s algorithm of Fig. 5 using \( \text{com} \) in (1) of Sect. 2.2.2. \( \text{com} \) in Fig. 8, and the consistency specification language \( \text{cat} \) which requires reasoning on candidate executions (see Sect. 11).

\[ \text{building the communication scenarios corresponding to the pythia triples for} \text{cat} \text{requires us building several candidate executions involving relations between accesses (i.e., read/write events) as follows (we illustrate on case 1 of Fig. 10).} \]

- read-from \( \text{rf} \); for each pythia triple, we depict the read-from relation \( \text{rf} \) in red; for example for \( \text{rf} \{ \text{F2}_1, \ 0, \ false \} \), we create a read-from relation between the initial write of \( \text{false} \) to the variable \( \text{F2} \) at line 0 and the read of \( \text{F2} \) from line 4, at the \( i \)th iteration.
- program order \( \text{po} \); we also depict the program order edges between the accesses which are either the source or the target of a communication edge (viz., read-from and coherence). In case 1 of Fig. 10, the po edges in purple are between the lines 1 and 4 on process P0, and lines 10 and 13 on process P0. po is irreflexive and transitive (not represented on Fig. 10).

\[ \begin{align*}
\text{at} \ 7 & \land \ \text{at} \ 16 \land \text{com} \\
& \Rightarrow (-\text{F2}_4\text{end} \lor T_{14}^\text{end} = 1) \land (-\text{F1}_3\text{end} \lor T_{14}^\text{end} = 16) \\
\end{align*} \]

\( (i.e., \text{the invariant} \text{com} \text{holds at lines} 7 \text{and} 16) \) \[ \Rightarrow -\text{com} \]

\( \text{since by taking} \ i = \text{end} \text{and} \ j = \text{end}, \ \text{we have} \]

\( (\text{F2}_4\text{end} \lor T_{14}^\text{end} = 1) \land (\text{F1}_3\text{end} \lor T_{14}^\text{end} = 12) \)

Note that this calculation of \( \text{com} \) from the specification \( \text{com} \) and the anarchic inductive invariant \( \text{com} \) provides a formal method to discover \( \text{com} \) by calculational design. \( \text{com} \) is sufficient but also necessary, hence the weakest communication hypothesis, since for each possible case of communication excluded by \( \text{com} \), it is possible to find a counter-example execution of Peterson violating mutual exclusion (see Fig. 7 and 10).
The cat specification introduces additional relations between events.

- the coherence co (defined as with co from AllCo): we depict the coherence edges co relative to the variables that are mentioned by the pythia triples, in our case F1, F2 and T; see in case 1 of Fig. 10 the co edge in blue between the write of F1 in the prelude at line 0 and the write of F1 at line 1.
- the from-read fr (defined as fr = rf^{-1}, co): we depict in brown the edges from a read relative to a variable x that is mentioned by the pythia triples at all the writes following after the write referred by this read. For example in case 1 of Fig. 10 where the read r[1] R1 F2 at line 4 is from the initial write 0: w] F1 0 and the fr relation shows that write 10:w] F2 true comes later.
- The cut relations linking events in different processes that may appear on the same cut during a program execution.

Moreover, we depict relations that might not be directly expressible in cat such as in cases 6 and 7 of Fig. 10:

- the events e, giving a semantics to instructions; Events e are partitioned into the set W of writes (including initial writes IW), the set R of reads, F of fences, B of tests;
- the program order po, relating accesses in their order of execution (which is the program order in the original LISA program);
- the read-from rf describing a communication between a write and a read event;

The language provides additional basic built-in semantics bricks:

- the relation loc relating events accessing the same variable;
- the relation ext relating events from different processes;
- operators over relations, such as intersection & union |, inverse of a relation ~-1, sequence of relations ;, transitive closure +, cartesian product *; set difference \.

The cat user can define new relations using let or with...from..., and declare constraints over relations r, such irreflexive r and acyclic r (i.e., irreflexive r+).

**Sequential consistency in cat** Fig. 11 gives a definition of Sequential Consistency (SC) in cat. The intuition is that if e1 po acyclic po | rf | co | fr as sc e2 then event e1 should appear on a cut before that of e2 (since instructions are executed in program order), if w rf then the write event w should appear on a cut before that of the read event r (since it is only possible to read from past writes), if w co w' then the write event w should appear on a cut before that of the write event w' (since the write events hit the shared memory in the coherence order co), and if r fr w then the write event w should appear on a cut after that of the read event r (since otherwise the read event r has not read from the last write). If the events do not appear in this prescribed order, there will be a cycle in the disjunction of these relations, which is disallowed by acyclic.

Lamport SC (Lamport 1979) is defined by imposing that cuts on anarchic executions (see Fig. 2) must satisfy the requirements illustrated in Fig. 12 that is (12a) no read on a cut can read from a write on a later cut and (12b) a read on a cut from a write on a previous cut must be from the last previous cut with such a write. Theorem SC is SC (Alglave et al. 2010, 2012, Th. 3) shows that

**Figure 10:** Communication scenarios violating Scom for Peterson
In Fig. 10, each case has a reason written underneath for being rejected. This is for example a reflexive sequence that our cat specification Hcom will forbid. Before detailing how we write Hcom in cat, we give a glimpse of the cat language.

- The cat language (Alglave et al. 2016) is a domain specific language to describe consistency models succinctly by constraining an abstraction of program executions into a candidate execution (e, po, rf, IW) providing

- No prophecy beyond cuts
- No prophecy beyond cuts
- No prophecy beyond cuts
- No prophecy beyond cuts
- No prophecy beyond cuts
- No prophecy beyond cuts
- No prophecy beyond cuts

Figure 12: Lamport SC
Lamport’s SC implies SC by cat. However, there can be executions with cuts defined by SC by cat that is disallowed by Lamport’s SC. An example is case 7 in Fig. 10 where executing P1 and next P0 is accepted in both cases while entering simultaneously both critical sections is forbidden by Lamport’s SC but allowed by SC by cat (the reason being that in both cases terminated executions have the same abstraction as candidate executions).

- Defining the consistency specification Hcom. For each case in Fig. 10, we forbid a reflexive sequence. In cat, the specification given in Fig. 13.

- Proving that all the behaviours allowed by Hcom are allowed by Scom is done contrapositively i.e., \( \neg S_{com} \Rightarrow \neg H_{com} \). By \( \neg S_{com} \) in (1), we get \( \exists i, j \cdot \langle r[iF_1], 0, false \rangle \lor \langle r[iF_2], 17, false \rangle \lor \langle r[jT], 11, 1 \rangle \lor \langle r[iF_1], 8, false \rangle \lor \langle r[jT_1], 2, 2 \rangle \rangle \) which we put in disjunctive normal form and give the cases illustrated in Fig. 10, thus proving \( \neg H_{com} \).
are explicitly removed from the TSO model. Proving that all the behaviours allowed by $M = M_i, i \in [1, n]$ are allowed by $H_\text{com}$ is done by reductio ad absurdum in the semantic domain of the consistency specification language (e.g., candidate executions for $\text{cat}$). Suppose an execution of Peterson that is forbidden by $H_\text{com}$ yet allowed by $M$. By definition of $H_\text{com}$ in Fig. 13, there are 5 cases. Each of these cases may be forbidden by the WCM $M$ (e.g., SC) or prevented by adding fences (e.g., TSO).

- **When $M$ is SC.** In $\text{cat}$ speak, SC is modelled as given in Fig. 11. Now, all 4 sequences required to be irreflexive by $H_\text{com}$ are included in the transitive closure of $po | rf | co | fr$, and rejected on SC. Moreover, Lamport’s SC has no prophecies on cuts, thus excluding cases 6 and 7 in Fig. 10.

- **Adding labelled fences (in case of no prophecy beyond cuts).** Some WCMs (like those weaker than TSO) authorize the reordering of write and read events on different shared variables against the program order $po$. In this case, the restrictions of $H_\text{com}$ in Fig. 13 are not satisfied and Peterson is incorrect. In the case where there are no prophecies on cuts, the solution is to add labelled fences as shown in Fig. 14.

```
| 0  | 2 |
| f[br] | 1 |
| f[br] | 2 |
| (which can be placed anywhere in the process, e.g., before the second label). The specification of the fence is |
| let fre = (rf-1 | co | ext) |
| let rfe = rf & ext |
| let fence = fromto(tag2events('br')) |
| irreflexive fre; fence | fre; fence |
| irreflexive co; fence | fence; fence |
```

**Figure 14:** Labelled fences for Peterson

In the invariance proof, fences are $\text{skip}$ so the proof is unchanged. The fence semantics must be defined by a $\text{cat}$ specification ($P$ is the set of fence events) and $H_\text{com}$ strengthened as shown in Fig. 14. This implies the consistency specification of $H_\text{com}$ for Peterson algorithm in Fig. 13 since fence $\subseteq po$.

- **When $M$ is TSO.** In $\text{cat}$ speak, TSO is modeled as given in Fig. 15 (omitting the no-prophecy beyond cuts satisfied by TSO). The difference with SC in Fig. 11 is that $w po r$ is required if the

```
| 0  | 2 |
| f[cut] | CS1 |
| f[cut] | CS2 |
```

The specification of the synchronisation marker is (see Fig. 12a)

```
let cut = (tag2events('cut') | tag2events('cut')) |
irreflexive rf; po | cut; po |
```

**Figure 15:** TSO in cat

Thus certain executions forbidden by our specification $H_\text{com}$ of Peterson (see Fig. 13) will not be forbidden by the TSO model given in Fig. 15. Indeed all the executions that contain a sequence $fr; po; fr; po$ forbidden by our specification of Peterson involves a pair write-read in program order. These write-read pairs are explicitly removed from the $\text{tso}$ acyclicity check of Fig. 15, thus will not contribute to executions forbidden by the model. It is therefore necessary to implement the fences of Fig. 14. The first one between $\{1\}/\{2\}$ and $\{10\}/\{11\}$ is implemented naturally in TSO since write-write pairs cannot be reordered with respect to $po$. The second labelled fence between $2/4$ and $11/15$ can be implemented by $m\text{fence}$ in $x86$.

- **In presence of prophecy beyond cuts,** e.g., in LISA, implementing a spinlock where the busy waiting can anticipate the lock release is incorrect. So we introduce a synchronisation marker at the beginning of both critical sections, as shown in Fig. 16, to prevent such prophecies beyond cuts.

```
| 0  | 2 |
| f[cut] | CS1 |
| f[cut] | CS2 |
```

The specification of the synchronisation marker is (see Fig. 12a)

```
let cut = (tag2events('cut') | tag2events('cut')) |
irreflexive rf; po | cut; po |
```

**Figure 16:** Anti-prophecy synchronisation markers for Peterson

## 3. Related Works on Invariance Proof for WCM

Contrary to our approach, previous attempts to generalise the (Owicki and Gries 1976) invariance proof method from SC to WCMs are not parameterised by a formal specification of the WCM. Our formal specification of the WCM parameter takes the form of program-specific programmer-specified communication assertion $S_\text{com}$ shown to be implied by a program-specific programmer-specified consistency specification $H_\text{com}$ (e.g., in $\text{cat}$) itself implied by an architectural consistency specification $M$ (e.g., (Shasha and Smir 1988; Alglave 2010; Alglave et al. 2016)). These constraints $S_\text{com}$ hence $H_\text{com}$ are on communications only, in contrast to constraints on the execution order and the visibility of writes (Crary and Sullivan 2015) or the ordering between commands of (Bornat et al. 2015).

Our invariance proof method deals with WCMs without getting back to the world of SC. This is in contrast to previous methods exposing the store buffers in the program states (e.g., (Dan et al. 2015)) or explicitly considering all possible shuffles e.g. by program transformation (e.g., (Atig et al. 2011; Alglave et al. 2013; Miné 2012)).

In the classical (Turing 1949; Naur 1966; Floyd 1967; Hoare 1967) invariance proof method, (shared) variable names are used in proofs to denote the value of the program variables. This is a severe restriction for previous invariance proof methods since in WCM there is no notion of global time hence of “the” instantaneous value of a shared variable. We solve the problem using pythia variables, based on the idea that the value of a shared variable is locally known when a read is satisfied. Pythia variables are loosely akin to the “fresh variables” used in the semantics and implementation of Prolog (Cousot et al. 2009). They differ from ghost variables used for behavior-preserving instrumentation of programs with non-physical resources and from prophecy variables for backward reasonings (Abadi and Lamport 1991; Cousot 1981). They are used in the herd7 tool (Alglave and Maranget 2015).

The literature sticks to SC with communicated value naming by shared variable names through restrictions on the considered algorithm, programming language, assertion, and/or memory model. Specific restrictions on the considered algorithm are concurrent stacks (Dodds et al. 2015), Read-Copy-Update (RCU) implementation of linked lists (Tassarotti et al. 2015).

Specific restrictions on the considered programming language with a specific memory model include ARM machine-code (Myreen et al. 2007; Myreen and Gordon 2007) or a specific programming discipline such as data-race-free programs for causal memory (e.g., (Ahamad et al. 1995; Owens 2010)), total store order with store
buffer forwarding (e.g., Cohen and Schirmer 2010), or coherent causal memory (e.g., Cohen 2014). Specific restrictions on the considered assertions mostly involve some form of abstraction (Dinsdale-Young et al. 2010; Battt et al. 2013).

Finally, the most common restriction is on the considered specific WCM (e.g., the release-acquire fragment of the C11 memory model (Norris and Densmy 2013; Vafeiadis and Narayan 2013; Turon et al. 2014; Lahav and Vafeiadis 2015; Tassarotti et al. 2015; Doku and Vafeiadis 2016; Lahav et al. 2016) (Turon et al. 2014) “also considers isolation / ownership transfer properties”). TSO/PSO/RMO in (Buchhardt and Musuvathi 2008; Atig et al. 2010; Wehrman and Berdine 2011; Szaiczkowski et al. 2015), the Java Memory Model in (Klebanov 2004), a hierarchical memory model in (Barthe et al. 2008), causal consistency in (Gotsman et al. 2008), causal memory (e.g., (Cohen and Schirmer 2010)), or coherent buffer forwarding (e.g., (Cohen 2014)). (Dinsdale-Young et al. 2010; Batty et al. 2013). (Cohen 2014)).

We don’t have any of the above restrictions since the WCM is a parameter of our proof method and defines the communication relation if explicitly appearing in invariants.

4. The LISA Language and its Analytic Semantics

We present here the LISA language (Litmus Instruction Set Architecture) (Alglave and Cousot 2016). Its vocation is purely pedagogical at the moment, with an ambition to be quite minimal. It is supported by the herd7 tool (Alglave and Maranget 2015). To illustrate this section we will use Peterson’s algorithm in Fig. 5.

4.1 Syntax

LISA programs \( P = \{ p_{\text{start}} \} | P_0 \ldots | P_{n-1} \) on shared variables \( x \in \text{loc}[P] \) contain:

- a prelude \( p_{\text{start}} \) assigning initial values to shared variables (0 (false) by default). In the case of Peterson algorithm in Fig. 5, the prelude at line 0 assigns the value false to both variables \( P_1 \) and \( P_2 \), and the value 0 to T.
- processes \( P_0 \ldots P_{n-1} \) in parallel; each process:
  - has an identifier \( p \in \text{PIL} \) of program. Moreover, instructions can bear tags \( \text{true} \) (to model for example C++ release and acquire annotations).
- has a local registers (e.g., \( R_0 \), \( R_1 \)); registers are assumed to be different from one process to the next; if not we make them different by affixing the process identifier like so: \( (P_0:R) \);
- is a sequence of instructions.

Instructions can be:

- register instructions \( \text{mov} R_1 \) operation, where the operation has the shape \( op \ R \) r-value:
  - the operator \( op \) is arithmetic (e.g., add, sub, mult) or boolean (e.g., eq, neq, gt, ge);
  - \( R_1 \) and \( R_2 \) are local registers;
  - r-value is either a local register or a constant;
- read instructions \( x[s] \) \( R \) xinitiate the reading of the value of the
  - read instructions \( x[s] \) \( R \) xinitiate the reading of the value of the
  - write instructions \( w[x] \) \( R \) winitiate the writing of the value of the
  - branch instructions \( b[x] \) \( R \) binitiate the branch label \( b \) if the
  - fence instructions \( f[x] \) \( R \) fonly applies between instructions in the first set and instructions in the second set. The semantics of the fence \( f \) as applied to the instructions at \( l_1 \ldots l_m \) and \( l_{m+1} \ldots l_n \) is to be defined in a cat specification;
  - read-modify-write instructions (RMW) \( \text{rmw} [s] \) \( R \) \( (\text{reg-instrs}) \ x \) reads shared variable \( x \) into local register \( R \), the register instruc-
  -...
and Owicki-Gries’ name \( x \) the value of the shared variable \( x \), but we cannot use the same idea in the context of weak consistency models. Instead we name \( x_0 \) the value of shared variable \( x \) read at local time \( \theta \).

The events \( \tau_p \) on a trace \( \tau_p \) of process \( p \) are as follows (e.g. Fig. 10):

- register events: \( a(p, \ell, \text{mov } R, \theta, v) \);
- read events: \( r(p, \ell, \ell [x] R \times x, \theta, v) \);
- write events: \( w(p, \ell, \ell [x] x \times r-value, \theta, v) \);
- branch events are of two kinds:
  - \( l(p, \ell, b [x] \text{ operation } l, \theta) \) for the true branch;
  - \( l(p, \ell, b [x] \text{ operation } l, \theta) \) for the false branch;
- fence events: \( m(p, \ell, \ell [x] \{ l_1, \ldots, l_n \} \{ l_1', \ldots, l_n' \}, \theta) \);
- RMW events are of two kinds:
  - begin event: \( m(p, \ell, \text{beginrmw} [x] x, \theta) \);
  - end event: \( m(p, \ell, \text{endrmw} [x] x, \theta) \).

**States** \( \sigma = s(\ell, \theta, \rho, \nu, v) \) of a process \( p \) mention:

- \( \ell \), the current control label of process \( p \) (we have done \([P]\langle p \rangle \ell \) which is true if and only if \( \ell \) is the last label of process \( p \) which is written when process \( p \) does terminate);
- \( \theta \) is the stamp of the state in process \( p \);
- \( \rho \) is an environment mapping the local registers \( R \) of process \( p \) to their ground or symbolic value \( B(p) \);
- \( \nu \) is a valuation mapping the pythia variables \( x_0 \in \mathcal{P}(p) \) of a process \( p \) to their ground or symbolic value \( v(x_0) \). This is a partial map since the pythia variables \( i.e. \), the domain \( \text{dom}(\nu) \) of the valuation \( \nu \) augment as communications unravel. Values can be ground, or symbolic expressions over pythia variables.

The prelude process has no state (represented by \( \bullet \)).

**Sequence of cuts** A cut of a computation \( \xi = \tau_{\text{start}} \times \bigcap_{p \in \mathcal{P}} \tau_p \) is a tuple \( \bigcap_{p \in \mathcal{P}} (\tau_p \bullet \tau'_p) \) of pairs of events and states of trace \( \tau_p, p \in \mathcal{P} \). A cut records the point each process has reached in its computation. A well-formed sequence of cuts records an interleaving of computation steps.

### 4.2.2 Well-formedness conditions

We specify our anarchic semantics by the means of well-formedness conditions over executions \( \xi = \bigcap_{\ell} \times \bigcap_{\ell} \times \bigcap_{\ell} \).

**Conditions over computations** \( \tau_p = \tau_{\text{start}} \times \bigcap_{p \in \mathcal{P}} \tau_p \) are as follows:

- **Start**: traces \( \tau \) must all start with a unique false start event \( \tau_{\text{start}} \).
- **Uniqueness**: the events of states must be unique:
  - \( \forall q \in \mathcal{P} \cup \{\text{start}\} \) \( \forall i \in [0, 1 \mid |\tau_p|] \) \( \exists i \neq \tau_{\text{state}} \).
- **Initialisation**: all shared variables \( x \) are initialised once and only once to a value \( v_i \) in the prelude (or to \( v_0 = 0 \) by default).
- **Maximality**: a finite trace \( \tau_p \) of a process \( p \) must be maximal i.e., must describe a process whose execution is finished. Note that infinite traces are maximal by definition, hence need not be included in the following maximality condition:
  - \( \exists \ell \theta v. \tau_{\ell \theta v} = s(\ell, \theta, v) \bigwedge \) done \([P]\langle p \rangle \ell \).

**Conditions over the cut sequence** \( \pi \) of a computation \( \xi = \tau_{\text{start}} \times \bigcap_{p \in \mathcal{P}} \tau_p \) are as follows:

- **Start**: The initial cut \( \pi_0 = \prod_{p \in \mathcal{P}} (\tau_{\text{start}} \times \bigcap_{p \in \mathcal{P}} \tau_0 \).

---

**Step**: But for the final cut, if any, the next cut follows from a computation step. If \( \pi_i = \prod_{p \in \mathcal{P}} (\tau_{k_i+1} \times \bigcap_{p \in \mathcal{P}} \tau_0) \), \( k_i < |\tau_0| \) then \( \exists q \in \mathcal{P} \bigcup \{\text{stop}\} \) such that \( k_i + 1 < |\tau_0| \) and

\[
\pi_{i+1} = \prod_{p \in \mathcal{P}} (\tau_{k_i+1} \times \bigcap_{p \in \mathcal{P}} \tau_0) \bigwedge Wf_7(\xi)
\]

**Conditions over the communications** \( \mathcal{R} \) are as follows:

- \( \text{Satisfaction: a read event has at least one corresponding communication in } \mathcal{R} \):
  - \( Wf_8(\xi) \)

- \( \text{Singleness: a read event in the trace } \tau_p \text{ must have at most one corresponding communication in } \mathcal{R} \):
  - \( Wf_9(\xi) \)

Note however that a read instruction can be repeated in a program loop and may give rise to several executions of this instruction, each recorded by a unique read event.

**Match**: if a read reads from a write, then the variables read and written must be the same:

\[
[\text{rd} \langle p, \ell, x := v, \theta, v \rangle, \tau_{\ell \theta v} \langle p, \ell, x := v', \theta, v' \rangle] \in \mathcal{R} \Rightarrow (x = x') \text{.} \quad Wf_{10}(\xi)
\]

**Inception**: no communication is possible without the occurrence of both the read and (maybe initial) write it involves:

\[
[\text{wt} \langle w, r \rangle, \ell | \theta \rangle, \tau_{\ell \theta v} \langle w, r \rangle] \in \mathcal{R} \Rightarrow (\exists \in \mathcal{P} \bigcup \mathcal{Q} \bigcup \{\text{start}\} \text{.} \quad Wf_{11}(\xi)
\]

**Language-dependent conditions for LISA** are as follows:

- **Start**: the initial state of a trace \( \tau_p \) should be of the form:
  - \( \tau_{\text{st}} = s(l_0, \text{inf}_p, \lambda, \text{R} \times 0, \theta) \) \( Wf_{12}(\xi) \)

Where \( l_0 \) is the entry label of process \( p \) and \( \text{inf}_p \) is a minimal stack of \( p \).

- **Next state**: if at point \( k \) of a trace \( \tau_p \) of process \( p \), an execution \( \xi = \tau_{\text{start}} \times \bigcap_{p \in \mathcal{P}} \tau_p \times \pi \times \mathcal{R} \) the computation is in state \( \tau_{\text{st}} = s(\ell, \theta, v, \nu, \rho) \), then:
  - the next event must be generated by the instruction \( \text{instr} \triangleq \text{instr} \bigwedge \text{done}_p \langle p \rangle \ell \text{ at label } \ell \text{ of process } p \).
  - the next event has the form \( \tau_{\text{st}} = s(\ell, \theta, v, \nu, \rho) \).
  - the next state \( \tau_{\text{st}} = s(\ell, \theta, v, \nu, \rho) \).

**Formally:**

\[
[\text{wt} \langle \ell, \theta, v, \nu, \rho, \nu \rangle, \langle \ell, \theta, v, \nu, \rho, \nu \rangle] \in \mathcal{R} \Rightarrow (x = x') \text{.} \quad Wf_{11}(\xi)
\]

**We** give the form of the next event \( \tau_{\text{st}} \) for each LISA instruction:

- **Fence**\( \langle \text{instr} \triangleq \ell : [x] \{l_1, \ldots, l_n\} \{l_1', \ldots, l_n'\}; \ell' \ldots \rangle \text{.} \quad Wf_{12}(\xi)
\]

- **Register**\( \langle \text{instr} \triangleq \ell : \text{mov } R, \theta, \ell' \ldots \rangle \text{.} \quad Wf_{14}(\xi)
\]
4.2.3 Anarchic semantics

\[ \forall (p, \ell, x) \in X \times \ell \times \pi \times \sigma \]

\[ \text{read (instr } = \ell : r(x))_1 \text{ : forarchic begin instr = beginrmw[x] end event instr = endrmw[x] } x) : \]

\[ \text{rmw (instr } = \text{rmw}[x] \text{ x ) : for the begin instr = beginrmw[x] and end instr = endrmw[x] )} \]

\[ \text{of a program } P \subseteq \gamma (P) \text{ equivalently } \alpha_n(S[P]) \subseteq P \text{ i.e. } S[P] \subseteq P . \]

\[ \text{The invariance abstraction } \alpha_{inv}(P), P \in \phi(\Xi) \text{ on Fig. 17 collects states on all cuts of all traces at each control point of each process.} \]

\[ \alpha_{inv}(P) \triangleq \prod_{0 \leq n \leq 1} \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \subseteq \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \]

\[ \alpha_{inv}(P) \triangleq \prod_{0 \leq n \leq 1} \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \subseteq \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \]

\[ \text{An invariance property } \ell \in \Xi \text{, in particular the strongest invariance } \alpha_{inv}(S[P]) \text{ in } J, \text{ attatches a local invariance property } \ell_{\text{inv}}(P) \text{ at each program point } \ell \text{ of each process } p \text{ which is a relation between the process state and the state of all other processes (including their control state) on all cuts of all executions going through point } \ell \text{ of process } p . \]

\[ \text{of a program } P \subseteq \gamma (P) \text{ equivalently } \alpha_n(S[P]) \subseteq P \text{ i.e. } S[P] \subseteq P . \]

\[ \text{The invariance abstraction } \alpha_{inv}(P), P \in \phi(\Xi) \text{ on Fig. 17 collects states on all cuts of all traces at each control point of each process.} \]

\[ \alpha_{inv}(P) \triangleq \prod_{0 \leq n \leq 1} \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \subseteq \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \]

\[ \text{An invariance property } \ell \in \Xi \text{, in particular the strongest invariance } \alpha_{inv}(S[P]) \text{ in } J, \text{ attatches a local invariance property } \ell_{\text{inv}}(P) \text{ at each program point } \ell \text{ of each process } p \text{ which is a relation between the process state and the state of all other processes (including their control state) on all cuts of all executions going through point } \ell \text{ of process } p . \]

\[ \text{of a program } P \subseteq \gamma (P) \text{ equivalently } \alpha_n(S[P]) \subseteq P \text{ i.e. } S[P] \subseteq P . \]

\[ \alpha_{inv}(P) \triangleq \prod_{0 \leq n \leq 1} \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \subseteq \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \]

\[ \text{An invariance property } \ell \in \Xi \text{, in particular the strongest invariance } \alpha_{inv}(S[P]) \text{ in } J, \text{ attatches a local invariance property } \ell_{\text{inv}}(P) \text{ at each program point } \ell \text{ of each process } p \text{ which is a relation between the process state and the state of all other processes (including their control state) on all cuts of all executions going through point } \ell \text{ of process } p . \]

\[ \text{of a program } P \subseteq \gamma (P) \text{ equivalently } \alpha_n(S[P]) \subseteq P \text{ i.e. } S[P] \subseteq P . \]

\[ \alpha_{inv}(P) \triangleq \prod_{0 \leq n \leq 1} \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \subseteq \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \]

\[ \text{An invariance property } \ell \in \Xi \text{, in particular the strongest invariance } \alpha_{inv}(S[P]) \text{ in } J, \text{ attatches a local invariance property } \ell_{\text{inv}}(P) \text{ at each program point } \ell \text{ of each process } p \text{ which is a relation between the process state and the state of all other processes (including their control state) on all cuts of all executions going through point } \ell \text{ of process } p . \]

\[ \text{of a program } P \subseteq \gamma (P) \text{ equivalently } \alpha_n(S[P]) \subseteq P \text{ i.e. } S[P] \subseteq P . \]

\[ \alpha_{inv}(P) \triangleq \prod_{0 \leq n \leq 1} \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \subseteq \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \]

\[ \text{An invariance property } \ell \in \Xi \text{, in particular the strongest invariance } \alpha_{inv}(S[P]) \text{ in } J, \text{ attatches a local invariance property } \ell_{\text{inv}}(P) \text{ at each program point } \ell \text{ of each process } p \text{ which is a relation between the process state and the state of all other processes (including their control state) on all cuts of all executions going through point } \ell \text{ of process } p . \]

\[ \text{of a program } P \subseteq \gamma (P) \text{ equivalently } \alpha_n(S[P]) \subseteq P \text{ i.e. } S[P] \subseteq P . \]

\[ \alpha_{inv}(P) \triangleq \prod_{0 \leq n \leq 1} \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \subseteq \bigcap_{\tau \in \Xi} (\gamma_\ell(S[P])) \]

\[ \text{An invariance property } \ell \in \Xi \text{, in particular the strongest invariance } \alpha_{inv}(S[P]) \text{ in } J, \text{ attatches a local invariance property } \ell_{\text{inv}}(P) \text{ at each program point } \ell \text{ of each process } p \text{ which is a relation between the process state and the state of all other processes (including their control state) on all cuts of all executions going through point } \ell \text{ of process } p . \]

\[ \text{of a program } P \subseteq \gamma (P) \text{ equivalently } \alpha_n(S[P]) \subseteq P \text{ i.e. } S[P] \subseteq P . \]
For completeness, we choose to describe exactly the communication conditions as follows: $S_{\text{com}} = \alpha_{\text{com}}(\alpha_{\text{com}}[H_{\text{com}}(\{S^{[P]}[P]\})])$. By defining the conditional invariance proof $S_{\text{com}} \Rightarrow S_{\text{com}}$ to be $\alpha_{\text{com}}[S_{\text{com}}(\{S^{[P]}[P]\}) \subseteq S_{\text{com}}(\{S^{[P]}[P]\})]$ and the inclusion proof $H_{\text{com}} \Rightarrow S_{\text{com}}$ to be $\alpha_{\text{com}}[H_{\text{com}}(\{S^{[P]}[P]\}) \subseteq S_{\text{com}}(\{S^{[P]}[P]\})]$, this calculation justifies the decomposition of the correctness proof into an invariance proof and an inclusion proof using an intermediate communication specification $S_{\text{com}}$.

### 7. Conditional Invariance Verification Conditions

We now present our invariance verification conditions for proving $S_{\text{com}} \Rightarrow S_{\text{com}}$, i.e., the properties that the logical assertions at each program point must satisfy to qualify as inductive invariants $S_{\text{com}}$.

#### 7.1 Pre, Post and Communication Conditions

For each of our verification conditions, we need to define general shapes of assertions; more specifically, we have:

- **Precondition**
  
  $$\text{PRE}_{\ell} \equiv S_{\text{com}}(\ell)[\kappa_0 \leftarrow \kappa, \theta_0, \rho_0, \nu_0, \tau_0, \text{rf}] \land S_{\text{com}}(\kappa_0)[\kappa_0 \leftarrow \ell, \theta_0, \rho_0, \nu_0, \tau_0, \text{rf}]$$

- **Postcondition**
  
  $$\text{POST}_{\ell} \equiv S_{\text{com}}(\ell)[\kappa_0 \leftarrow \kappa', \theta_0, \rho_0, \nu_0, \tau_0, \text{rf}]$$

- **Communication Condition**
  
  $$\text{COM}_{\ell} \equiv S_{\text{com}}(\ell)[\text{rf}] \land S_{\text{com}}(\ell)[\text{rf}]$$

meaning that a read or write instruction at $\ell$ of process $p$ may execute (since the invariant $S_{\text{com}}(\ell)[\text{rf}]$ holds) and communicate according to $\text{rf}$ (as specified by $S_{\text{com}}(\ell)[\text{rf}]$).

#### 7.2 Initialisation Verification Condition

For each process $p$, the invariant at the entry point $\ell_0^p$ must be true when the other processes are also at their entry, with all registers initialised to 0 and no pythia variable:

- **Precondition**
  
  $$\text{PRE}_p \equiv \bigwedge_{\ell \in \ell_0^p} [\kappa_0 \leftarrow \ell, \theta_0, \text{inf}_p, \rho_0, \nu_0, \text{rf} \leftarrow \lambda R \cdot 0, \nu_0, \text{rf} \leftarrow \emptyset]$$

- **Postcondition**
  
  $$\text{POST}_p \equiv \bigwedge_{\ell \in \ell_0^p} [\kappa_0 \leftarrow \ell, \theta_0, \text{inf}_p, \rho_0, \nu_0, \text{rf} \leftarrow \lambda R \cdot 0, \nu_0, \text{rf} \leftarrow \emptyset]$$

#### 7.3 Sequential Verification Conditions

The verification conditions for the sequential proof require to prove that if the precondition inductive invariant $\text{PRE}_p$ holds at point $\ell$ of process $p$ and the instruction at label $\ell$ is executed and goes to $\ell'$ and the communication is allowed by specification $S_{\text{com}}$ then the postcondition inductive invariant $\text{POST}_p$ holds at point $\ell'$ with the updated stamp, environment and valuation. The sequential verification conditions are the special case of the non-interference ones given below for $\text{PRE}_{\ell'} = \text{PRE}_{\ell'} \land \text{POST}_{\ell'} = \text{POST}_{\ell'}$.

#### 7.4 Non-interference Verification Conditions

The verification conditions for the non-interference proof require to prove that if the precondition inductive invariant $\text{PRE}_{\ell'}$ holds at point $\ell$ of process $p$ and any other process $r$ executes an instruction $\kappa : \text{inst} \kappa'$ at label $\kappa'$, goes to $\kappa''$, and the communication is allowed by specification $S_{\text{com}}$, then the postcondition inductive invariant $\text{POST}_{\ell'r}$ at point $\ell$ still holds with the updated stamp, environment and valuation.

- **For local side-effect free marker instructions $\kappa$:**
  
  $$\text{inst} \kappa'$$

  where $\text{inst} \equiv [\ell \mid \ell' \mid \ell'']$, $w_{\text{rf}}(\ell, x) \leftarrow \text{rf}$$

  beginvarx[n] x, endvarx[n] x:

  (marker)

  $$\text{PRE}_{\ell} \Rightarrow \text{POST}_{\ell'} \cdot$$

  (for the register assignment $\kappa$ : $\text{movR operation} \kappa'$ ; (assign)

  $$\text{PRE}_{\ell}[\rho_r, \nu_r] \Rightarrow \text{POST}_{\ell'}[\rho_r, \nu_r]$$

  (for a read instruction $\kappa$ : $\text{r}x[n] \leftarrow \text{rf}$$

  $$\text{PRE}_{\ell}[\theta_r, \rho_r, \nu_r, \text{rf}] \land \text{rf}\{w(q, \ell', w[n] \leftarrow \text{r}x[n] \leftarrow \text{rf} \leftarrow \text{r}x[n] \leftarrow \text{r}x[n]), \theta_r, \rho_r, \nu_r, \text{rf}\} \in \text{rf}$$

  (for a test instruction $\kappa$ : $\text{b}x$ operation $l_1^{\kappa'}$ ; (test)

  $$\text{PRE}_{\ell}[\rho_r, \nu_r] \land \text{sat}(\text{E}[\text{operation}](\rho_r, \nu_r)) \Rightarrow \text{POST}_{\ell'}[\rho_r, \nu_r]$$

  (where $\text{sat}$ checks for satisfiability of symbolic boolean expressions or for truth of ground values. These formal non-interference verification conditions can be rephrased as inference rules (in an informal style “$\{P\}_i S_i \{Q\}_j$, $i \in [1, n]$ are interference free” (Owicki and Gries 1976)).

#### 7.5 Communication Verification Conditions

Assertions associated with read and write instructions must satisfy certain sanity conditions that stem from the semantics:

- All process read instructions $\ell$ : $\text{r}x[n] \leftarrow \text{rf}$ must read either from an initial or a reachable program write, allowed by the communication hypothesis $\exists P[x_1, \ldots, x_m]$ (where all free variables in predicate $P$ but $x_1, \ldots, x_m$ are existentially quantified):

  $$\text{COM}_{\ell}[\theta_q, \rho_r] \land \text{rf} \mid \exists P[w(q, \ell, w[n] \leftarrow \text{r}x[n] \leftarrow \text{rf} \leftarrow \text{r}x[n]), w[n] \leftarrow \text{r}x[n]) \in \text{rf}$$

  (satisfaction)

- A read event can read from only one write event:

  $$\text{COM}_{\ell}[\theta_q, \rho_r, \nu_r, \text{rf}] \Rightarrow \nu_r = \text{rf}$$

- The values $\nu$ allowed to be read by the communication hypothesis must originate from reachable program write instructions $\ell$ : $\text{w}x[n] \leftarrow \text{rf}$:

  $$\forall \text{rf} \mid \exists P[w(q, \ell, w[n] \leftarrow \text{r}x[n] \leftarrow \text{rf} \leftarrow \text{r}x[n]), w[n] \leftarrow \text{r}x[n]) \in \text{rf}$$

  (match)

- The inceptive condition $\text{Wf}_{\ell_1}(\ell')$ is not required since non-existent communications can only lead to more imprecise invariants, which is sound. However, it is always possible to take inpection into account to get precise invariants for completeness.

The communications taken into account in $\text{rf}$ must include all those of the anarchic semantics as restricted by $S_{\text{com}}$ (by Sect. 10 and Sect. 11) and the imprecision can only be on communicated values (including in absence of inception).

**Example (Thin air 1)** In absence of loops, stamps are the unique program labels. We write $\text{rf}(x_{\ell_0}, \ell_0, \nu_0)$ for $\text{rf} \{w[q, \ell_0, w[n] \leftarrow \text{r}x[n] \leftarrow \text{rf} \leftarrow \text{r}x[n]), \theta_r, \rho_r, \nu_r, \text{rf}\}$ and define $\Gamma_1 \triangleq \{\text{rf}(x_{\ell_0}, 0, 0), \text{rf}(x_{\ell_1}, 7, 42)\}$, and $\text{rf}(y_{\ell_0}, 0, 0), \text{rf}(y_{\ell_1}, 3, 42)\}$.

- $\circ \leftarrow 0 \land \text{rf} \in \Gamma_1$:

  \begin{align*}
  \text{rf} & \in \circ \leftarrow 0 \land \text{rf} \in \Gamma_1; \quad \text{rf} \in \circ \leftarrow 0 \land \text{rf} \in \Gamma_1;
  \end{align*}

- $\text{rf}(x_{\ell_0} \leftarrow 0, 0) \land \text{rf}(x_{\ell_1} \leftarrow 42) \land \text{rf}(y_{\ell_0} \leftarrow 0, 0) \land \text{rf}(y_{\ell_1} \leftarrow 3, 42) \land \text{rf} \in \Gamma_1$.

By the communication proof for any $\text{rf} \in \Gamma_1$, communicated values cannot be different (match), $\text{rf}$ can neither be chosen to be a superset by (satisfaction) and (singleness) nor a subset (which is the subject of Sect. 10 and 11). For example, at 2, $x_1 \in \{0, 42\}$ is
prevented by the read rule, all communicated values are readable. Values must come from writes so \( x_1 \in \{0, 42, 43 \} \) at 2 is prevented by (match). In case of an unconditional branch \( b \) true 8; at 8, any \( r' x_1, 7, \ldots \) is \( r' \) prevented by (satisfaction) i.e., it is not possible to read from a non-reachable write.

8. Calculation of the Invariance Proof \( S_{\text{com}} \Rightarrow S_{\text{inv}}, \)

Soundness and Completeness by Design

The calculation for the invariance proof \( S_{\text{com}} \Rightarrow S_{\text{inv}}, \) formally \( \alpha_{\text{inv}}(S^p[\sharp]) \cap S_{\text{com}} \subseteq S_{\text{inv}}, \) goes on by induction on the length of trace prefixes, which requires the use of the inductive invariant \( S_{\text{inv}}. \)

The basis \( Wf_{12}(\xi) \) yields the initialisation condition. The sequential verification condition for \( S_{\text{com}}(\ell) \) is obtained when performing a computation step \( Wf_{13}(\xi), \ldots, Wf_{15}(\xi) \) in the current process \( p \) while the non-interference is obtained when performing a step in another process \( r \neq p. \) The communication proof requirements follow from \( Wf_{8}(\xi) \) to \( Wf_{11}(\xi). \)

This calculational design yields Th. 2 showing that the proposed proof method is sound (i.e., if the verification conditions are all satisfied then the invariance statement \( S_{\text{inv}} \) is true for the program anarchic semantics \( S^p[\sharp] \) with communications restricted by the specification \( S_{\text{com}}. \) Reciprocally, the proof method is complete so that if an invariance statement \( S_{\text{inv}} \) is true for the program anarchic semantics \( S^p[\sharp] \) with communications restricted by the specification \( S_{\text{com}}, \) then this can always be proved thanks to a stronger inductive invariant \( S_{\text{inv}} \) satisfying all verification conditions.

As usual the completeness proof provides no clue on how to choose the inductive invariant \( S_{\text{inv}} \) since it is based on the choice \( S_{\text{inv}} = \alpha_{\text{inv}}(S^p[\sharp]) \cap S_{\text{com}}, \) i.e., the exact abstraction of the semantics which in general is not computable.

The soundness and completeness proof is set-theoretical. In practice, one uses a logic with an interpretation, and so the soundness proof is identical using the interpretation of the logical formula. This is a problem however for the completeness proof since, in general, \( S_{\text{inv}} = \alpha_{\text{inv}}(S^p[\sharp]) \cap S_{\text{com}} \) cannot be expressed as a formula of the chosen logic. One can consider higher-order logics as in e.g. (Back and von Wright 1990) but they cannot be handled e.g. by SMT solvers. The usual restriction is to relative completeness under the assumption that \( \alpha_{\text{inv}}(S^p[\sharp]) \cap S_{\text{com}} \) is expressible in the logic (Cook 1978, 1981).

Theorem 2 (Invariance proof \( S_{\text{com}} \Rightarrow S_{\text{inv}} \) \( S_{\text{com}} \Rightarrow S_{\text{inv}}, \) formally \( \alpha_{\text{inv}}(S^p[\sharp]) \cap S_{\text{com}} \subseteq S_{\text{inv}}, \) if and only if there exists \( S_{\text{com}} \subseteq S_{\text{inv}}, \) which is inductive for \( P, \) i.e., satisfies the interpretation of the initialisation (7.2), sequential (7.3), non-interference (7.4), and communication (7.5) verification conditions of Sect. 7.

The following Th. 3 supports our claim that our invariance proof method for WCM is an extension of Lamport’s invariance proof method for sequential consistency.

Theorem 3 (Generalisation of Lamport proof method) The verification conditions of Th. 2 for the inductive invariant \( S_{\text{com}} \) reduce to (Lamport 1977) proof method for sequential consistency.

Our invariance proof method for WCM is also an extension of (Owicky and Gries 1976) for sequential consistency since, by the argument given in (Cosut and Courtois 1980), the auxiliary variables can always be chosen as local registers (simulating program counters) so auxiliary variables need not to be shared.

9. Calculation of the Inclusion Proof \( H_{\text{com}} \Rightarrow S_{\text{com}}, \)

Soundness and Completeness by Design

The calculation of the inclusion proof \( H_{\text{com}} \Rightarrow S_{\text{com}}, \) formally \( \alpha_{\text{com}}(\alpha_{\text{ana}}(H_{\text{com}})(S^p[\sharp])) \subseteq S_{\text{com}}, \) yields the verification conditions for the communication specification \( S_{\text{com}} \) in Th. 4 below. Define

\[ S^\text{ana}[H_{\text{com}}][P] = \alpha_{\text{ana}}[H_{\text{com}}][(S^p[\sharp])][\xi = \text{allowed}]. \]

We have

\[ \alpha_{\text{com}}(\alpha_{\text{ana}}(H_{\text{com}})(S^p[\sharp])) \subseteq S_{\text{com}} \]

\[ \Leftrightarrow \alpha_{\text{com}}(\alpha_{\text{ana}}(H_{\text{com}})(S^p[\sharp])) \subseteq S_{\text{com}} \]

\[ \{ \text{def. } \alpha_{\text{ana}}(H_{\text{com}})[P] \}

\[ \Leftrightarrow \forall \xi \in S^\text{ana}[H_{\text{com}}][P] \ . \ \alpha_{\text{com}}(\{ \xi \}) \subseteq S_{\text{com}} \]

\[ \{ \text{def. } \alpha_{\text{ana}} \text{ preserves } \cup \}

\[ \Leftrightarrow \forall \xi \in S^\text{ana}[H_{\text{com}}][P] \ . \ \bigcup_{p=1}^{n} \{ \alpha_{\text{com}}(\xi') \}(P(L)) \subseteq S_{\text{com}}(P(L)) \]

\[ \{ \text{def. of } \alpha_{\text{com}} \text{ and } S_{\text{com}}(P) \text{ so that } \xi \text{ has the form } \xi = \tau_{\text{start}} \times \prod_{p=0}^{n-1} T_{p} \times \tau_{\text{step}} \times \tau_{\text{r}} \} \]

\[ \Leftrightarrow \forall \xi \in S^\text{ana}[H_{\text{com}}][P] \ . \ \bigcup_{p=0}^{n-1} \{ \alpha_{\text{com}}(\xi') \}(P(L)) \subseteq S_{\text{com}}(P(L)) \]

\[ \{ \text{def. of } \alpha_{\text{ana}} \text{ and } S_{\text{com}} \}

\[ \text{So we have proved}

Theorem 4 (Inclusion proof) The verification conditions (20) are sound and complete for proving the inclusion \( H_{\text{com}} \Rightarrow S_{\text{com}}. \)

Observe that the completeness proof of Th. 4 assumes that \( H_{\text{com}} \Rightarrow S_{\text{com}}. \) If the consistency specification language is not expressive enough there might be no way to express a strong enough consistency specification \( H_{\text{com}}, \) a source of incompleteness. This is the case e.g. of cat designed to describe architectures so that e.g. memory values are abstracted away which may not be the case in \( S_{\text{com}}. \) This means that the design of the program \( P \) must ensure that \( S_{\text{com}} \) is weak enough to be implementable.

Observe also that Th. 4 requires analyzing all possible executions of the program, which is seldom feasible. Moreover, this is in contradiction with the idea of invariance proof which purpose is precisely to avoid to reason directly on program executions. We explore an alternative inclusion proof method \( H_{\text{com}} \Rightarrow S_{\text{com}} \) using an anarchic invariant i.e., an invariant of the anarchic semantics.

10. Anarchic Invariant

An anarchic invariant \( S_{\text{ana}}^p \) of the anarchic semantics \( S^p[\sharp] \) is an invariant that takes into account all possible communications allowed by the program semantics (programmers would say the program logic). A general invariant is not enough. The problem is that a general invariant can be of the form “if the communications satisfy given hypotheses then the computations satisfy an invariant property”. Obviously this is an invariant of the anarchic semantics but since not all possible communications allowed by the program semantics are characterized by the invariant, this is not an anarchic invariant.

The following Th. 5 shows how to find an anarchic invariant of the anarchic semantics using our proof method with the guarantee that no hypothesis has been made on the communications (but for those disallowed by the semantics as in e.g. \( [x] \) \( R+R+1 \); \( [y] \) \( R1 \) \( x \) with no feasible execution on \( z \).)

The anarchic semantics \( S^p[\sharp] \) considers all possible write events \( W_P(\theta_q, q) \ . \ \theta_q \in \{ \text{start} \} \) \( \theta_q \in \{ \theta_q \} \)

\[ W(\theta_q, q) \triangleq \{ \text{prefix } \{ \theta_q, q \} \times r-value, \} \]

\[ \text{in } \{ \text{expr } \{ \theta_q, q \} \times r-value, \} \]

\[ \{ \theta_q \} \in \{ \text{expr } \{ \theta_q, q \} \times r-value, \} \]
and all possible read events $\overline{R}[P] (\theta_p, x), p \in \Phi$, $\theta_p \in \Sigma(p)$.

$\overline{R}[P] (\theta_p, x) \triangleq \{ (p, \ell_p, r [x] \in \theta_p, x_p) \mid \ell_p \in L(p) \land \text{instr} [P] \ell_p = r [x] \}$

The anarchic communications $rf \in \overline{R}[P]$ are between matching write and read events for any shared variable $x \in \text{loc} [P]$.

$\overline{P}[P] \triangleq \{ \{ rf w [\omega_q, r, v] \mid x \in \text{loc} [P] \land q \in \Phi \cup \{ \text{start} \} \land \theta_q \in \Sigma(q) \land p \in \Phi \land \theta_p \in \Sigma(p) \} \mid \forall q, \theta_q, p, \theta_p, v, w \}$

Define an anarchic invariant to be

$S^n_w \triangleq \bigcup \{ \text{inv} [rf] \mid rf \in \overline{P}[P] \land \text{inv} [rf] \text{ is inductive for } P \}$

$S^n_w$ is an invariant of the anarchic semantics $S^n_w [P]$ (i.e., $\alpha_{\text{ana}} (S^n_w [P]) \subseteq S^n_w$) if and only if there exists an anarchic invariant $S^n_w$ such that $S^n_w \subseteq S^n_w$.

$S^n_w$ can be obtained by considering the values $v_{0}, v_{1}$ to be symbolic, applying the sequential, non-interference, and communication verification conditions of Th. 2, and solving the resulting constraints in the symbolic variables $v_{0}, v_{1}$.

Example 2 (Thin air 2) Consider the following process:

$P_1 \mid P_2 \mid P_3 \mid P_4 \mid P_5$

The possible anarchic communications are

$\overline{P}[P] \subseteq \{ \{ rf (x_1, 0, v_0) \mid v_0 \in \mathbb{D} \} \cup \{ rf (x_1, 3, v_2) \mid v_2 \in \mathbb{D} \} \cup \{ rf (x_2, 4, v_3) \mid v_3 \in \mathbb{D} \} \mid 0 \in \{ x = 0 \} \}$

Define

$\text{com}_0 \triangleq x_0 \land rf = \{ rf (x_1, 0, v_0) \}$

$\text{com}_1 \triangleq x_1 \land rf = \{ rf (x_1, 3, v_2) \}$

$\text{com}_2 \triangleq x_2 \land rf = \{ rf (x_2, 4, v_3) \}$

$\text{com}_3 \triangleq x_3 \land rf = \{ rf (x_1, 0, v_0) \}$

$\text{com}_4 \triangleq x_4 \land rf = \{ rf (x_1, 3, v_2) \}$

$\text{com}_5 \triangleq x_5 \land rf = \{ rf (x_2, 4, v_3) \}$

where $v_0, v_2, v_3$ are fresh symbolic variables.

The following invariant (including the terms outlined in red).

0: \{ $x = 0$ \} \{\}
1: \{ $R_1 = 0 \land (\text{com}_0 \lor \text{com}_4)$ \} \{ R [] R1 x1 \{ ~x1 \} \}
2: \{ $R_1 = x_1 \land (\text{com}_0 \lor \text{com}_4)$ \}

if $R_1 = 42$ then
3: \{ $R_1 = x_1 \land (\text{com}_0 \lor \text{com}_4)$ \} \{ w[] x1 \}
4: \{ $R_1 = x_1 \land (\text{com}_0 \lor \text{com}_4)$ \} \{ w[] x2 \}
5: \{ $R_1 = x_1 \land (\text{com}_0 \lor \text{com}_4)$ \} \{ f[] ; \}
6: \{ $R_1 = x_1 \land (\text{com}_0 \lor \text{com}_4)$ \}

By the (match) rule at $v_0 = 0$, at $x_3 = 41$, and at $v_4 = 42$.

We get the anarchic inductive invariant $S^n_w$ (excluding the infeasible communications of the terms outlined in red which are false according to the program semantics).

0: \{ $x = 0$ \} \{\}
1: \{ $R_1 = 0 \land (x_1 = 0 \land rf = \{ rf (x_1, 0, 0) \}) \lor (x_1 = 42 \land rf = \{ rf (x_1, 4, 42) \})$ \} \{ R [] R1 x1 \{ ~x1 \} \}
2: \{ $R_1 = x_1 \land (x_1 = 0 \land rf = \{ rf (x_1, 0, 0) \}) \lor (x_1 = 42 \land rf = \{ rf (x_1, 4, 42) \})$ \} \{ R1 = 42 \}

if $R_1 = 42$ then
3: \{ $R_1 = 42 \land rf = \{ rf (x_1, 4, 42) \}$ \} \{ w[] x1 \}
4: \{ $R_1 = 42 \land rf = \{ rf (x_1, 4, 42) \}$ \} \{ w[] x2 \}
5: \{ $R_1 = 42 \land rf = \{ rf (x_1, 4, 42) \}$ \} \{ f[] ; \}
6: \{ $R_1 = 0 \land rf = \{ rf (x_1, 0, 0) \} \lor (R1 = 42 \land rf = \{ rf (x_1, 4, 42) \})$ \}

11. Candidate Executions for cat Specifications

The proof of $H_{\text{ana}} \Rightarrow S_{\text{ana}}$ by Th. 4 requires extracting executions from the semantics $H_{\text{ana}} ([H_{\text{ana}}] \circ \alpha_{\text{ana}} [S^n_w [P]])$. $\alpha_{\text{ana}} [S^n_w [P]] \circ \alpha_{\text{ana}} [S^n_w [P]]$ i.e., candidate executions $\alpha_{\text{ana}} (S^n_w [P])$ for cat specifications. Th. 6 shows that it is sound to extract a super-set of the anarchic candidate executions $\alpha_{\text{ana}} (S^n_w [P])$ directly from an anarchic invariant $S^n_w$ (excluding any possible communication) and the program $P$. Notice that a general invariant would not do since some possible communications could be omitted, hence the insistence on the use of an anarchic invariant $S^n_w$ (i.e., satisfying the invariance verification conditions of Sect. 7 with $S_{\text{ana}} = \text{true}$).

Moreover, by the completeness result, the extraction can be exact by choosing a precise enough anarchic invariant.

Theorem 5 (Anarchic invariant) $\text{S}_{\text{ana}}$ is an invariant of the anarchic semantics $\alpha_{\text{ana}} (S^n_w [P])$ if and only if $\alpha_{\text{ana}} (S^n_w [P]) \subseteq S^n_w$.

The set of candidate executions defined by an assertion $A$ at point $\ell$ of process $P$ is $\alpha_{\text{ana}} (S^n_w, A) \triangleq \{ \alpha_{\text{ana}} (A, rf') \mid rf' \in C_P(\ell) \}$

and the possible communication relations are $C_P(\ell) \triangleq \{ (r, f') \mid 3 \in \Phi, \ell \land (\text{com} \lor \text{com} r' \lor \text{com} f') \}$

By the (match) rule at $v_0 = 0$, at $x_3 = 41$, and at $v_4 = 42$.

For all $p \in \Phi$ and $\ell \in L(p)$, the extraction $\alpha_{\text{ana}} (S^n_w, A)$ from an anarchic invariant $S^n_w$ is sound and complete.

Considering local process invariants, we get Cor. 7 ensuring all possible anarchic communications allowed by the program semantics are taken into account when proving $S^n_w$.

Corollary 7 (Candidate executions out of local anarchic invariants) For all $p \in \Phi$ and $\ell \in L(p)$, the extraction $\alpha_{\text{ana}} (S^n_w, A)$ of the program candidate executions from an anarchic invariant $S^n_w$ (i.e., $\alpha_{\text{ana}} (S^n_w [P]) \subseteq S^n_w$) is sound (i.e., $\alpha_{\text{ana}} (S^n_w, A)$).

12. Inclusion Proof Revisited for cat Specs

To prove $H_{\text{ana}} \Rightarrow S_{\text{ana}}$ without reasoning directly on the anarchic semantics $\alpha_{\text{ana}} (S^n_w [P])$, we have to consider (a superset of) all feasible communications, which by Th. 1 determine (a superset of) all feasible executions. Extracting these communications from an anarchic invariant $S^n_w$ by def. $C(S^n_w)$, we can, by Th. 6 and Cor. 7 and for each of these communication relations, extract (an approximation of) the corresponding candidate execution (which can be as precise as necessary by completeness) on which the semantics of the cat language can be applied to check whether the candidate execution, essentially the communication relation, is consistent with the WCM defined by $H_{\text{ana}}$. If this is the case, it should be accepted by the communication specification invariant $S^n_w$. It follows that $S^n_w$ does not forget any feasible (allowed by the program semantics/logic) and consistent (allowed by $H_{\text{ana}}$) communication relation, hence by Th. 1 execution.

Theorem 8 (Inclusion proof for cat) $H_{\text{ana}} \Rightarrow S_{\text{ana}}$ (formally $\alpha_{\text{ana}} (H_{\text{ana}}) \circ \alpha_{\text{ana}} (S^n_w [P]) \subseteq S^n_w$ and only if there exists an anarchic invariant $S^n_w$ satisfying the following (inclusion) verification condition):
∀p ∈ P. ∀ℓ ∈ L(p). ∀rf′ ∈ C.p,ℓ(S_{a_p}(ℓ)) . (inclusion) 
\left( \exists\omega[H_{\text{com}}(\omega_2(S_{a_p}(ℓ), rf)) \land S_{a_p}(ℓ)(rf \leftrightarrow rf')] \Rightarrow S_{a_p}(ℓ)(rf \leftrightarrow rf') \right) .

The cat specification $H_{\text{com}}$ is a conjunction of conditions on a candidate execution $(e, po, rf, iw)$ of the form

\begin{align*}
\text{let } r &= R(e, po, rf, iw) \\
\text{acyclic } [\text{irreflexive } | \text{empty } | \text{not empty } r]
\end{align*}

where the relation $r \in \varphi(e \times e)$ is a function of $e$, $po$, $rf$, and $iw$, as defined by the cat language semantics \( \exists \omega[H_{\text{com}}] \) (Alglave et al. 2016). We have acyclic($r$) if and only if irreflexive($r$) so we only have to handle irreflexivity and emptiness.

The proof of $S_{a_p}(ℓ)(rf) \Rightarrow \neg \exists\omega[H_{\text{com}}(\omega_2(S_{a_p}(ℓ), rf))]$ is by contraposition, i.e., any communication rejected by $S_{a_p}$ is also rejected by $H_{\text{com}}$. Assuming $\neg S_{a_p}(ℓ)(rf)$, the check $\neg \exists\omega[H_{\text{com}}(\omega_2(S_{a_p}(ℓ), rf))]$ considers each of these reflexivity or emptiness conditions in turn.

13. A Proof of PostgreSQL

The PostgreSQL example 1 of Fig. 21 was considered in (Alglave et al. 2013) for bounded bug-finding on a multi-core PowerPC system. We prove, under appropriate hypotheses, the critical section (CS) specification $S_{\text{cs}} \triangleq \neg(\exists\mathbf{t} \in [\mathbb{R}] \land \exists\mathbf{a} \in \mathbb{R})$ plus non-starvation (CSSs are entered infinitely often).

**Anarchic communications of PostgreSQL.** The anarchic communications $\Gamma$ of Fig. 21. We write $r\forall(x, \langle : \theta, \forall \rangle)$ to state that the read into pythia variable $x \forall$ was from an write event marked $\forall \theta$ of value $\forall v$ generated by the action at process label $\forall$. The markers $\forall \theta \theta' \theta$ are the vectors of iteration counters of the loops enclosing the read/write instruction. We write $r\forall\forall\theta\forall\theta'\forall\theta"$ for the possible read-froms of variable latch0 ($=L$) or variable flag0 ($=F$) in process $P$, $\forall \in \{0,1\}$ for unique stamp $\forall$ (as encoded by loop counters). All possible cases are considered in Fig. 21.

All possible communications are obtained by considering that each read of a shared variable in the loops can read from any initial, past or future write to this variable, a different choice being possible at each read. So each $\Gamma \in \Gamma$, $\Gamma = \{r\forall\forall\theta\forall\theta'\forall\theta" | \forall \in \mathbb{N} \land \forall \in \{0,1\} \land \forall \in \{0,1\} \land \forall \in \{0,1\} \}$ encodes a particular read-from relation $rf$ specifying that for the $i^{th}$ iteration in the external loop 1–12 and the $j^{th}$ iteration in the internal loop 1–4 of process $P$, 3: $r\forall\forall\theta\forall\theta'\forall\theta"$ will read as specified by $r\forall\forall\theta\forall\theta'\forall\theta$ where $\forall \in R\forall\forall\theta\forall\theta'\forall\theta$ while 6: $r\forall\forall\theta\forall\theta'\forall\theta"$ will read as specified by $r\forall\forall\theta\forall\theta'\forall\theta"$.

**Anarchic inductive invariant of PostgreSQL.** The inductive invariant $S_{\text{out}}$ is given in Fig. 21. It depends on $\Gamma$ encoding a communication $rf$. $S_{\text{out}}$ assumes that $\Gamma$ belongs to an unspecified set $\Gamma$ of possible communications (this dependency is written $S_{\text{out}}(\Gamma, \Gamma)$). So $S_{\text{out}}(\Gamma, \Gamma)$ is valid under the communication hypothesis $S_{\text{out}}(\Gamma, \Gamma) \triangleq (\Gamma \in \Gamma)$. It follows that $S_{\text{out}}(\Gamma, \Gamma)$ is an inductive anarchic invariant.

**Necessary and sufficient communication specification $S_{\text{com}}$ for mutual exclusion.** We derive in Fig. 19 the communication specification $S_{\text{com}}$ in Fig. 18 by calculational design from the critical section requirements.

It follows that $(S_{\text{com}}(\Gamma, \Gamma) \land S_{\text{out}}(\Gamma, \Gamma)) \Rightarrow S_{\text{com}}(\Gamma, \Gamma)$ so $S_{\text{com}}(\Gamma, \Gamma) \Rightarrow S_{\text{com}}(\Gamma, \Gamma)$ since $S_{\text{com}}(\Gamma, \Gamma) \Rightarrow S_{\text{out}}(\Gamma, \Gamma)$, proving mutual exclusion under the $S_{\text{com}}$ communication hypothesis.

The proof that $S_{\text{com}}(\Gamma, \Gamma)$ is also necessary is done by providing counter-examples. For example, a candidate execution counter-example to $S_{\text{com}}$ is given in Fig. 20 (where the control points of the cut of the traces where the error occurs (i.e., both processes are simultaneously in their critical section) is marked).
This LISA fence can be implemented for SC (fences = po), TSO (fences = tso), and ARM (fences = ad), which is the scenario of Fig. 20. This is prevented using LISA fences with the following cat specification.

with co from AllCo
  let flu = fencerel(Flw)
  let fencerel(S) = (pos(S); po & fromto(S))
  let fcs = deps | flw
  let deps = fencerel(Fdep k (S * _))
  let fr = r'f^-1; co
  let fri = F & tagdevents('fdep')
  let cyc = fencerel(F'('flw')

1. Either rf[0], (0; 0) ∈ Γ ∧ F0 = 0, which is the scenario of Fig. 20. This is prevented using LISA fences with the following cat specification.

2. Or, 0 ∈ N, rf[0] = (0; i8, 0), which is impossible by the (satification) verification condition.

14. Conclusion and Perspectives

We introduced a new style of analytic specification of parallel program semantics decomposed into (a) an anarchic semantics with true parallelism and separate communications and (b) consistency requirements on communications (e.g. with labelled synchronization markers in cat).

This leads to a new style of invariance specification for arbitrary weak consistency models using pythia variables and a communication relation to denote communicated variables. This allows us to consider arbitrary subsets of the anarchic communications hence of the anarchic executions.

By calculational design using an invariance abstraction of the analytic semantics, we designed a new sound and complete conditional invariance proof method (where the communication hypotheses are expressed as an invariant) and a new sound and complete inclusion proof method (to prove the communication hypotheses valid for a consistency requirement e.g. in cat).

This leads to a new way of designing portable concurrent programs, by porting programs and their proofs from one architecture to another with different architectural specifications by renumbering.

Other proof methods can be derived by applying the proof method transformations of (Cousot and Cousot 1982) e.g. backward reasoning or by reductio ad absurdum. Alternative, more modular, proof methods can be designed in the same vein, such as rely-guarantee (Coleman and Jones 2007; Miné 2014), Inductive Data Flow Graphs to suppress irrelevant details of an invariance proof (Farzan et al. 2013), or Separation logic (O’Hearn 2007).

References

Figure 18: Communication specification for PostgreSQL

\[ (\neg S_{\text{comm}}(\Gamma, \Gamma)) \land S_{\text{comm}}(\Gamma, \Gamma) \]

\[ \equiv at(8) \land at(28) \land S_{\text{comm}}(\Gamma, \Gamma) \]

\[ \land \{ \text{def. invariance specification } S_{\text{invar}} \} \]

\[ \Rightarrow at(8) \land at(28) \land (\exists i, k, \ell, n, e, \ell_0, e_0, k_0, e_0) \land \Gamma \in \Gamma \land (r1R^0_1[\ell] \land \neg r1R^0_1[\ell] \land r1R^1_1[\ell] \land \neg r1R^1_1[\ell]) \]

\[ \{ \text{by invariant specification } S_{\text{invar}}(\Gamma, \Gamma) \} \]

\[ \Rightarrow at(8) \land at(28) \land (\exists i, k_1, k, \ell, e_0, k_0, e_0, k_0, e_0) \land \Gamma \in \Gamma \land (\neg r1R^0_1[\ell] \land \neg r1R^0_1[\ell]) \land r1R^1_1[\ell]) \]

\[ \{ \text{def. } r1R^0_1[\ell], r1R^0_1[\ell], r1R^1_1[\ell], r1R^1_1[\ell] \} \land r1R^1_1[\ell] \]

\[ \{ r2g(\ell, \ell', \ell, \Gamma), \neg r2g(\ell, \ell', \ell, \Gamma) \} \]

\[ \{ \text{implies that } x_2 = v, A \land (B \lor C) = (A \land B) \lor (A \land C), \exists \text{ does not distribute over } \lor, \Gamma \land (A \times B) = \exists \times \Gamma \} \]

\[ \Rightarrow at(8) \land at(28) \land (\neg S_{\text{comm}}(\Gamma, \Gamma) \lor \neg S_{\text{comm}}(\Gamma, \Gamma)) \]

\[ \Rightarrow \neg S_{\text{comm}}(\Gamma, \Gamma) \]

by defining \( S_{\text{comm}}(\Gamma, \Gamma) \equiv at(8) \land at(28) \land S_{\text{comm}}(\Gamma, \Gamma) \land S_{\text{comm}}(\Gamma, \Gamma) \land S_{\text{comm}}(\Gamma, \Gamma) \land \) and the \( S_{\text{comm}}(\Gamma, \Gamma) \) as in Fig. 18.

Figure 19: Calculational design of \( S_{\text{comm}} \) for PostgreSQL

Figure 20: Counter-examples to PostgreSQL with WCM
\{0: \text{latch0} = 0; \text{flag0} = 0; \text{latch1} = 1; \text{flag1} = 1; \}

1: \{ \Gamma \in \Gamma \}
   \text{do } \{i\}
2: \{ \Gamma \in \Gamma \}
   \text{do } \{i\}
3: \{ \Gamma \in \Gamma \}
   \text{while } (\text{Rl0}=0) \{k_i\}
4: \{ \Gamma \in \Gamma \} \land \text{Rl0} = L0_i^j \land (\text{r0Rl0}^j_i [\Gamma] \lor \text{r1Rl0}^j_i [\Gamma])
   \text{while } (\text{Rl1}=0) \{n_i\}
5: \{ \Gamma \in \Gamma \} \land \text{r1Rl0}^j_i [\Gamma]
   \text{w}[] \text{latch0} 0
6: \{ \Gamma \in \Gamma \} \land \text{r1Rl0}^j_i [\Gamma]
   \text{w}[] \text{flag0} \{\sim F0^i\}
7: \{ \Gamma \in \Gamma \} \land \text{r1Rl0}^j_i [\Gamma] \land \text{Rf0} = F0^i
   \land (\text{r0Rl0}^i_j [\Gamma] \lor \text{r1Rl0}^i_j [\Gamma])
   \text{if } (\text{Rf0} \neq 0) \text{ then}
8: \{ \Gamma \in \Gamma \} \land \text{r1Rl0}^j_i [\Gamma] \land \text{r1Rl0}^i_j [\Gamma]
   (* \text{critical section} *)
   \text{w}[] \text{flag0} 0
9: \{ \Gamma \in \Gamma \} \land \text{r1Rl0}^j_i [\Gamma] \land \text{r1Rl0}^i_j [\Gamma]
   \text{w}[] \text{flag1} 1
10: \{ \Gamma \in \Gamma \} \land \text{r1Rl0}^j_i [\Gamma] \land \text{r1Rl0}^i_j [\Gamma]
   \text{w}[] \text{latch1} 1
11: \{ \Gamma \in \Gamma \} \land \text{r1Rl0}^j_i [\Gamma] \land \text{r1Rl0}^i_j [\Gamma]
   \text{fi}
12: \{ \Gamma \in \Gamma \}
   \text{while true}
13: (false)
21: \{ \Gamma \in \Gamma \}
   \text{do } \{\ell\}
22: \{ \Gamma \in \Gamma \}
   \text{do } \{m_i\}
23: \{ \Gamma \in \Gamma \}
   \text{r}[] \text{Rl1} \text{latch1} \{\sim L1^\ell_i\}
24: \{ \Gamma \in \Gamma \} \land \text{Rl1} = L1^\ell_i \land (\text{r0Rl1}^\ell_i [\Gamma] \lor \text{r1Rl1}^\ell_i [\Gamma])
   \text{while } (\text{Rl1}=0) \{n_i\}
25: \{ \Gamma \in \Gamma \} \land \text{r1Rl1}^\ell_i [\Gamma]
   \text{w}[] \text{latch1} 0
26: \{ \Gamma \in \Gamma \} \land \text{r1Rl1}^\ell_i [\Gamma]
   \text{r}[] \text{Rf1} \text{flag1} \{\sim F1^\ell\}
27: \{ \Gamma \in \Gamma \} \land \text{r1Rl1}^\ell_i [\Gamma] \land \text{Rf1} = F1^\ell
   \land (\text{r0Rf1}^\ell [\Gamma] \lor \text{r1Rf1}^\ell [\Gamma])
   \text{if } (\text{Rf1} \neq 0) \text{ then}
28: \{ \Gamma \in \Gamma \} \land \text{r1Rl1}^\ell_i [\Gamma] \land \text{r1Rf1}^\ell [\Gamma]
   (* \text{critical section} *)
   \text{w}[] \text{flag1} 0
29: \{ \Gamma \in \Gamma \} \land \text{r1Rl1}^\ell_i [\Gamma] \land \text{r1Rf1}^\ell [\Gamma]
   \text{w}[] \text{flag0} 1
30: \{ \Gamma \in \Gamma \} \land \text{r1Rl1}^\ell_i [\Gamma] \land \text{r1Rf1}^\ell [\Gamma]
   \text{w}[] \text{latch0} 1
31: \{ \Gamma \in \Gamma \} \land \text{r1Rl1}^\ell_i [\Gamma] \land \text{r1Rf1}^\ell [\Gamma]
   \text{fi}
32: \{ \Gamma \in \Gamma \}
   \text{while true}
33: (false)

Invariants:
\begin{align*}
\text{r0Rl0}^j_i [\Gamma] & \triangleq (\text{rf}(L0_i^j, \langle 0:, \ldots, 0 \rangle) \in \Gamma \land L0_j^i = 0) \lor (\exists i_5 \in \mathbb{N} . \text{rf}(L0_j^i, \langle 5:, i_5, 0 \rangle) \in \Gamma \land L0_j^i = 0) \\
\text{r1Rl0}^j_i [\Gamma] & \triangleq (\exists a_30 \in \mathbb{N} . \text{rf}(L0_j^i, \langle 30:, a_30, 1 \rangle) \in \Gamma \land L0_j^i = 1) \\
\text{r0R0}^i_j [\Gamma] & \triangleq (\exists i_8 \in \mathbb{N} . \text{rf}(F0^i, \langle 8:, i_8, 0 \rangle) \in \Gamma \land F0^i = 0) \\
\text{r1R0}^i_j [\Gamma] & \triangleq (\exists a_29 \in \mathbb{N} . \text{rf}(F0^i, \langle 29:, a_29, 1 \rangle) \in \Gamma \land F0^i = 1) \\
\text{r0Rl1}^\ell_i [\Gamma] & \triangleq (\exists a_{25} \in \mathbb{N} . \text{rf}(L1_m^\ell, \langle 25:, a_{25}, 0 \rangle) \in \Gamma \land L1_m^\ell = 0) \\
\text{r1Rl1}^\ell_i [\Gamma] & \triangleq (\exists i_9 \in \mathbb{N} . \text{rf}(F1^\ell, \langle 9:, i_9, 1 \rangle) \in \Gamma \land F1^\ell = 1)
\end{align*}

Communications:
\begin{align*}
\text{RL0}^j_i & \triangleq (\text{rf}(L0_j^i, \langle 0:, \ldots, 0 \rangle), \text{rf}(L0_j^i, \langle 5:, i_5, 0 \rangle), \text{rf}(L0_j^i, \langle 30:, a_30, 1 \rangle) | i_5 \in \mathbb{N} \land a_30 \in \mathbb{N}) \\
\text{RF0}^i & \triangleq (\text{rf}(F0^i, \langle 0:, \ldots, 0 \rangle), \text{rf}(F0^i, \langle 8:, i_8, 0 \rangle), \text{rf}(F0^i, \langle 29:, a_29, 1 \rangle) | i_8 \in \mathbb{N} \land a_29 \in \mathbb{N}) \\
\text{RL1}^\ell_i & \triangleq (\text{rf}(L1_m^\ell, \langle a_{25}, 0 \rangle), \text{rf}(L1_m^\ell, \langle 25:, a_{25}, 0 \rangle), \text{rf}(L1_m^\ell, \langle i_9, i_9, 0 \rangle) | i_9 \in \mathbb{N} \land i_9 \in \mathbb{N}) \\
\text{RF1}^\ell & \triangleq (\text{rf}(F1^\ell, \langle 0:, \ldots, 1 \rangle), \text{rf}(F1^\ell, \langle 9:, i_9, 1 \rangle) | \Gamma \land F1^\ell = 1)
\end{align*}

Anarchic communications:
\begin{align*}
\Gamma = (\text{r0Rl0}^j_i, \text{r1Rl0}^j_i, \text{r0Rl1}^\ell_i | i \in \mathbb{N} \land j_i \in [0, k_i] \land \ell \in \mathbb{N} \land j \in [0, n_\ell]) | \forall i \in \mathbb{N} \land j_i \in [1, k_i] . \text{r0Rl0}^j_i \land \text{r0Rl0}^j_i \land \text{RF0}^i \land \forall \ell \in \mathbb{N} \land m_\ell \in [1, m_\ell] . \text{r1Rl1}^\ell_i \land \text{r1Rl1}^\ell_i \land \text{RF1}^\ell
\end{align*}

Figure 21: Inductive invariant $S_{ind}(\Gamma, \Gamma)$ of PostgreSQL (under hypothesis $S_{con}(\Gamma, \Gamma) \triangleq (\Gamma \in \Gamma), \Gamma \subseteq \overline{\Gamma}$)