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

Patrick Cousot
New York University, USA, emer. École Normale Supérieure, PSL.
pcousot@cims.nyu.edu, cousot@ens.fr

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 `cat` as an 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 (Owicki-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 2016; Grief 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 previous write (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 our ***invariance proof method*** as an abstraction of the analytic semantics. Thus our method is parameterised by a WCM.

POPL'17 January 18–20, 2017, Paris, France

DOI: http://dx.doi.org/10.1145/10.1145/3009837.3009883

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 `lb` in Fig. 1.

```
0:{ x = 0; y = 0; }
P0          || P1          ;
1:r[] r1 x  || 11:r[] r2 y;
2:w[] y 1   || 12:w[] x 1 ;
3:          || 13:        ;
```

Figure 1: `lb` algorithm in LISA

In `lb`, 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 line `12` for `P1`).

1.1 Anarchic Semantics

Fig. 2 gives one of the four anarchic executions of `lb`. After initial-

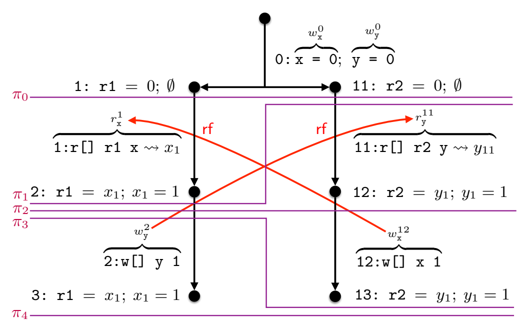


Figure 2: One anarchic execution for `lb`

ising `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. x_1 for the read $r^1_{\mathtt{x}}$ = `1:r[] r1 x`. 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 $r^1_{\mathtt{x}}$ takes its value from the write $w^{12}_{\mathtt{x}}$ (so the value 1 is assigned to the pythia variable x_1).

The interleaving of the processes' executions is given by *cuts*. The sequence of cuts $\pi_0; \pi_1; \pi_2; \pi_3; \pi_4$ in Fig. 2 formalises the interleaving `x=0; y=0; r[] r1 x; r[] r2 y; w[] x 1; w[] y 1.`

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 actions (defined by the cuts), all possible coherence orders are a priori possible (and will be considered in `cat` with `with co from AllCo` (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 `lb` of Fig. 2.

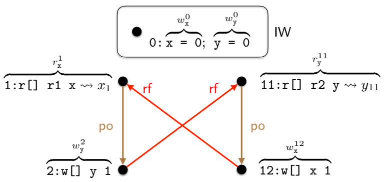


Figure 3: Candidate execution for `lb`

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 `lb` 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: $r^1_{\mathtt{x}}$ po $w^2_{\mathtt{y}}$ rf $r^{11}_{\mathtt{y}}$ po $w^{12}_{\mathtt{x}}$ rf $r^1_{\mathtt{x}}$.

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 $S_{com} \Rightarrow S_{inv}$ is invariant for `lb` where $S_{inv} = (\mathtt{at}\{3\} \wedge \mathtt{at}\{13\}) \Rightarrow \neg(\mathtt{r1} = 1 \wedge \mathtt{r2} = 1)$ and the communication hypothesis $S_{com} = \{\langle w^{12}_{\mathtt{x}}, r^1_{\mathtt{x}}\rangle, \langle w^2_{\mathtt{y}}, r^{11}_{\mathtt{y}}\rangle\} \not\in \mathsf{rf}$ 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 *Litmus Instruction Set Architecture (*LISA*)* language (Alglave and Cousot 2016) of the `herd7` tool (Alglave and Maranget 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:

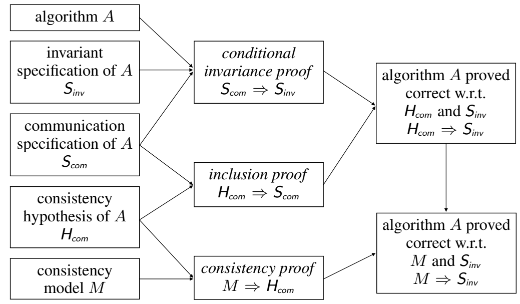


Figure 4: Our method

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                              || P1:
1:w[] F1 true                    || 10:w[] F2 true;
2:w[] T 2                        || 11:w[] T 1;
3:do {i}                         || 12:do {j}
4:   r[] R1 F2 {⇝ F2^i_4}        || 13:   r[] R3 F1; {⇝ F1^j_13}
5:   r[] R2 T  {⇝ T^i_5}         || 14:   r[] R4 T;  {⇝ T^j_14}
6:while R1 ∧ R2 ≠ 1 {i_end}      || 15:while R3 ∧ R4 ≠ 2; {j_end}
7:   (* CS1 *)                   || 16:   (* CS2 *)
8:w[] F1 false                   || 17:w[] F2 false;
9:                               || 18:
```

Figure 5: Peterson algorithm in LISA

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).

   We write A in LISA, using LISA's special fence *synchronisation markers* if needed, which allow to define in `cat` between which program points (perhaps sets of program points) synchronisation is needed for correctness;

2. Prove the correctness $S_{com} \Rightarrow S_{inv}$ 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. 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 ($\mathtt{i}_{\mathtt{end}}$ at line `6` and $\mathtt{j}_{\mathtt{end}}$ 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 ℓ in the program, reading the variable `x` and placing its result into register `R`, is

decorated with the pythia variable $\{\rightsquigarrow \mathtt{x}_\ell^n\}$, where n is the iteration counter (for nested loops we record all iteration counters of all surrounding loops).

2.1.2 Invariant specification S_{inv}

Fig. 6 gives an invariant specification of Peterson stating that both processes may not be simultaneously in their critical sections.

```
1: {true}        10: {true}
...              ...
7: {¬at{16}}     16: {¬at{7}}
...              ...
9: {true}        18: {true}
```

Figure 6: Invariant specification S_{inv} for Peterson's algorithm

2.2 Communication Specification S_{com}

The next step in our specification process consists in stating an invariant communication specification S_{com}, expressing which inter-process 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_{inv} of Fig. 6.

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

- on process `P0`, the read at line `4` reads the value that `F2` was initialised to, at line `0`, so that `R1` contains `false`. And, the read at line `5` reads from any write of `T`, so that `R2` contains one of the values `0`, `1`, or `2`, indifferently.
- on process `P1`, the read at line `13` reads from the initial value of `F1` so that `R3` contains `false`. The read at line `14` reads from `0`, `11`, or `2` so that `R4` 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_{inv}. Another erroneous behaviour is illustrated in Fig. 7b. The value of

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                  || P1:
1:w[] F1 true        || 10:w[] F2 true;
2:w[] T 2            || 11:w[] T 1;
3:do                 || 12:do
4:   r[] R1 F2       || 13:   r[] R3 F1;
5:   r[] R2 T        || 14:   r[] R4 T;
6:while R1 ∧ R2 ≠ 1  || 15:while R3 ∧ R4 ≠ 2;
7:   (* CS1 *)       || 16:   (* CS2 *)
8:w[] F1 false       || 17:w[] F2 false;
```

(a) Incorrect flags

```
0:{ w F1 false; w F2 false; w T 0; }
P0:                  || P1:
1:w[] F1 true        || 10:w[] F2 true;
2:w[] T 2            || 11:w[] T 1;
3:do                 || 12:do
4:   r[] R1 F2       || 13:   r[] R3 F1;
5:   r[] R2 T        || 14:   r[] R4 T;
6:while R1 ∧ R2 ≠ 1  || 15:while R3 ∧ R4 ≠ 2;
7:   (* CS1 *)       || 16:   (* CS2 *)
8:w[] F1 false       || 17:w[] F2 false;
```

(b) Incorrect turns

Figure 7: Incorrect executions of Peterson algorithm with WCM

`F2` and `F1` is indifferent. 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_{inv}.

2.2.2 Communication specification S_{com}

Let us express the communication scenarios depicted in Fig. 7 as an invariant. We write the pythia triple $rf\langle \mathtt{x}_\theta,\ \ell,\ v\rangle$ to mean that the read event $r = \ell'$: `r[] R x` $\{\rightsquigarrow \mathtt{x}_\theta\}$, or more precisely its unique pythia variable x_θ, takes its value v from evaluating the expression `e` of the write event $w = \ell$: `w[] x e` to v (so $\langle w,\ r\rangle \in \mathsf{rf}$ at ℓ and ℓ' and $x_\theta = v$ at ℓ'). (The communication verification conditions will check that $\langle w,\ r\rangle \in \mathsf{rf}$ and the local invariant at ℓ implies that $\mathtt{e} = v$.) We define our communication specification S_{com} as follows:

$$S_{com} \triangleq \neg[\exists i, j.[rf\langle \mathtt{F2}_4^i,\ \mathtt{0},\ \mathsf{false}\rangle \vee rf\langle \mathtt{F2}_4^i,\ \mathtt{17},\ \mathsf{false}\rangle \vee rf\langle \mathtt{T}_5^i,\ \mathtt{11},\ \mathtt{1}\rangle] \wedge [rf\langle \mathtt{F1}_{13}^j,\ \mathtt{0},\ \mathsf{false}\rangle \vee rf\langle \mathtt{F1}_{13}^j,\ \mathtt{8},\ \mathsf{false}\rangle \vee rf\langle \mathtt{T}_{14}^j,\ \mathtt{2},\ \mathtt{2}\rangle]] \tag{1}$$

S_{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 `P0` enters its critical section at the i^{th} iteration of its waiting loop (corresponding to the pythia variables $\mathtt{F2}_4^i$ and $\mathtt{T}_5^i$) because
  - either the read at line `4` and i^{th} iteration (corresponding to the pythia variable $\mathtt{F2}_4^i$) takes its value, `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 $\mathtt{T}_5^i$) takes its value, `1`, from the write at line `11`;
- And the second process `P1` enters its critical section at the j^{th} iteration of its waiting loop (corresponding to the pythia variables $\mathtt{F1}_{13}^j$ and $\mathtt{T}_{14}^j$) because
  - either the read at line `13` and j^{th} iteration (corresponding to the pythia variable $\mathtt{F1}_{13}^j$) takes its value, `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 $\mathtt{T}_{14}^j$) takes its value, `2`, from the write at line `2`.

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

2.3 Our Proof Method

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

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

We now detail each proof in turn.

2.3.1 Conditional invariance proof $S_{com} \Rightarrow S_{inv}$

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

1. invent a stronger invariant S_{ind}, which is inductive;
2. prove that S_{ind} 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_{com}; effectively we prove $S_{com} \Rightarrow S_{ind}$.
3. prove that S_{ind} is indeed stronger than S_{inv} (*i.e.*, $S_{ind} \Rightarrow S_{inv}$);

From $S_{com} \Rightarrow S_{ind}$ and $S_{ind} \Rightarrow S_{inv}$ we conclude that $S_{com} \Rightarrow S_{inv}$, which was our goal. We now illustrate the correctness proof method on Peterson.

• ***An inductive invariant*** S_{ind}***, stronger than*** S_{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 rf *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).

```
0: { w F1 false; w F2 false; w T 0; }
{F1=false ∧ F2=false ∧ T=0} }
1:  {R1=0 ∧ R2=0}
    w[] F1 true
2:  {R1=0 ∧ R2=0}
    w[] T 2
3:  {R1=0 ∧ R2=0}
    do {i}
4:  {(i=0 ∧ R1=0 ∧ R2=0) ∨
     (i>0 ∧ R1=F2^{i-1}_4 ∧ R2=T^{i-1}_5)}
        r[] R1 F2 {⇝ F2^i_4}
5:  {R1=F2^i_4 ∧ (i=0 ∧ R2=0) ∨
                 (i>0 ∧ R2=T^{i-1}_5)}
        r[] R2 T {⇝ T^i_5}
6:  {R1=F2^i_4 ∧ R2=T^i_5}
    while R1 ∧ R2≠1 {i_end}
7:  {¬F2^{i_end}_4 ∨ T^{i_end}_5=1}
    skip (* CS1 *)
8:  {¬F2^{i_end}_4 ∨ T^{i_end}_5=1}
    w[] F1 false
9:  {¬F2^{i_end}_4 ∨ T^{i_end}_5=1}

10: {R3=0 ∧ R4=0}
    w[] F2 true;
11: {R3=0 ∧ R4=0}
    w[] T 1;
12: {R3=0 ∧ R4=0}
    do {j}
13: {(j=0 ∧ R3=0 ∧ R4=0) ∨
     (j>0 ∧ R3=F1^{j-1}_13 ∧ R4=T^{j-1}_14)}
        r[] R3 F1 {⇝ F1^j_13};
14: {R3=F1^j_13 ∧ (j=0 ∧ R4=0) ∨
                  (j>0 ∧ R4=T^{j-1}_14)}
        r[] R4 T; {⇝ T^j_14}
15: {R3=F1^j_13 ∧ R4=T^j_14}
    while R3 ∧ R4≠2 {j_end} ;
16: {¬F1^{j_end}_13 ∨ T^{j_end}_14=2}
    skip (* CS2 *)
17: {¬F1^{j_end}_13 ∨ T^{j_end}_14=2}
    w[] F2 false;
18: {¬F1^{j_end}_13 ∨ T^{j_end}_14=2}
```

Figure 8: (Anarchic) invariants of Peterson algorithm

Following (Lamport 1977), we use program counters so we do not need (Owicki and Gries 1976)'s shared auxiliary variables. The equivalence proof of (Cousot and Cousot 1980) shows that the auxiliary variables in (Owicki and Gries 1976) can always be chosen as local variables (*i.e.*, registers in LISA) simulating program counters. This proof easily generalises to the WCM anarchic semantics. So we avoid the problem that "OG's auxiliary variables, in general, are unsound under weak memory because they can be used to record the exact thread interleavings and establish completeness under SC" (Lahav and Vafeiadis 2015). Our solution, using program counters or auxiliary registers is both sound and (relatively) complete, and simpler and more general that the ghost states of (Lahav and Vafeiadis 2015; Jung et al. 2016).

• S_{ind} ***is inductive under the hypothesis*** S_{com} is decomposed into an *initialisation proof* that the entry invariant is true, a *sequential proof* that the invariants hold when executing one process sequentially, a *non-interference proof* when running processes concurrently, and finally a *communication proof*.

The novelty of our approach is in the communication proof. We must prove that if an invariant is true at some process point ℓ of a process p and a `read` for x_θ is performed then the value received into x_θ is that of a matching write. Of course only the communications allowed by the communication invariant S_{com} and all of them have to be taken into account.

For Peterson, the invariants do not say anything on the value assigned to the pythia variables so that the invariants are true for any value carried by the pythia variables. More precisely, the read at line `4` can read from the writes at line `0`, `10` or `17`. The invariant at line `4` does not make any distinction on these cases and just states that some value $\texttt{F2}^i_4$ has been read and assigned to `R1`. Similarly the read of `T` at line `5` can read from the writes at line `0`, `2`, or `11` and the invariant just states that some value $\texttt{T}^i_5$ is read and assigned to `R2`.

The invariant of Fig. 8 holds for the anarchic semantics since no hypothesis is made on communications `rf` and therefore no possible communication has been forgotten.

• S_{ind} ***is stronger than*** S_{inv} ***under the hypothesis*** S_{com}***;*** On the Peterson example, the invariance proof does not make any use of the communication hypothesis S_{com}. It is however used in the *mutual exclusion proof*, that S_{ind} is stronger than S_{inv}. We prove that $(S_{com} \wedge S_{ind}) \Rightarrow S_{inv}$ or equivalently $(S_{ind} \wedge \neg S_{inv}) \Rightarrow \neg S_{com}$, as follows:

$$\texttt{at}\ \ 7 \wedge \texttt{at}\ \ 16 \wedge S_{ind}$$
$$\Rightarrow (\neg\texttt{F2}^{i_{\text{end}}}_4 \vee \texttt{T}^{i_{\text{end}}}_5 = 1) \wedge (\neg\texttt{F1}^{j_{\text{end}}}_{13} \vee \texttt{T}^{j_{\text{end}}}_{14} = 2)\}$$
$$\wr\, i.e., \text{ the invariant } S_{ind} \text{ holds at lines } 7 \text{ and } 16\, \wr$$
$$\Rightarrow \neg S_{com} \quad \wr \text{since by taking } i = i_{\text{end}} \text{ and } j = j_{\text{end}}, \text{ we have } (\texttt{F2}^i_4 = \mathsf{false} \vee \texttt{T}^i_5 = 1) \wedge (\texttt{F1}^j_{13} = \mathsf{false} \vee \texttt{T}^j_{14} = 2)\wr$$

Note that this calculation of S_{com} from the specification S_{inv} and the anarchic inductive invariant S_{ind} provides a formal method to discover S_{com} by calculational design. S_{com} is sufficient but also necessary, hence the weakest communication hypothesis, since for each possible case of communication excluded by S_{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 H_{com}

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

2.3.3 Inclusion proof $H_{com} \Rightarrow S_{com}$

As illustrated in Fig. 9, the WCMs H_{com} and M_i, $i \in [1, n]$ belong to the domain of consistency specifications (*e.g.* candidate executions for `cat`) while S_{com} belongs to the different domain of invariants. The proof $H_{com} \Rightarrow S_{com}$ must therefore be done in the most

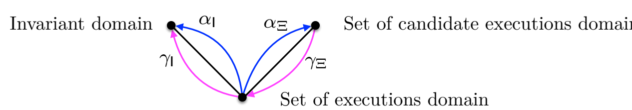


Figure 9: Hierarchy of abstractions

abstract domain more concrete than both of these domains, which is the semantic domain of sets of executions. The same way that we derived S_{com} from the program specification $\neg S_{inv}$, we derive H_{com} from the communication specification $\neg S_{com}$. This is an abstraction since *e.g.* in `cat` shared variable names and their values are abstracted away. So, in general, H_{com} will allow less behaviors that S_{com} and the M_i less that H_{com}. The proof $H_{com} \Rightarrow S_{com}$ proceeds as follows:

- we build the communication scenarios corresponding to the pythia triples given in the communication specification $\neg S_{com}$ from an anarchic invariant S^{a}_{inv};
- we write a consistency specification H_{com} (*e.g.* in `cat`) which will forbid each of these communication scenarios.

We illustrate the proof method with Peterson's algorithm of Fig. 5 using S_{com} in (1) of Sect. 2.2.2, S^{a}_{inv} in Fig. 8, and the consistency specification language `cat` which requires reasoning on candidate executions (see Sect. 11).

• ***Building the communication scenarios corresponding to the pythia triples*** for `cat` 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* `rf`: for each pythia triple, we depict the read-from relation `rf` in red; for example for $rf\langle \texttt{F2}^i_4,\ \texttt{0},\ \mathsf{false}\rangle$, we create a read-from relation between the initial write of `false` to the variable `F2` at line `0` and the read of `F2` from line `4`, at the i^{th} iteration.
- *program order* `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).

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 to all the writes to `x` coming after the write read by this read; For example in case 1 of Fig. 10 where the read `r[] R1 F2` of `F2` at line `4` is from the initial write `0: w[] F2 0` and the fr relation shows that write `10:w[] F2 true` comes later.

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

- the *cut* relations linking events in different processes that may appear on the same cut during a program execution.

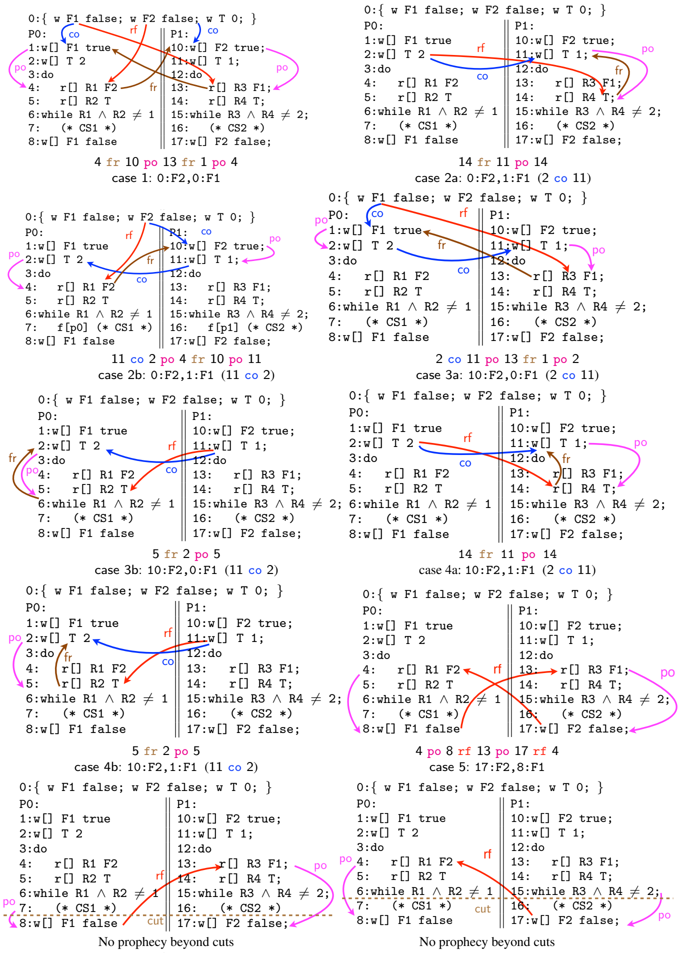


Figure 10: Communication scenarios violating S_{com} for Peterson

In Fig. 10, each case has a reason written underneath for being rejected. This is for example a reflexive sequence that our `cat` specification H_{com} will forbid. Before detailing how we write H_{com} 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

- *events* e, giving a semantics to instructions; Events e are partitionned 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`+`).

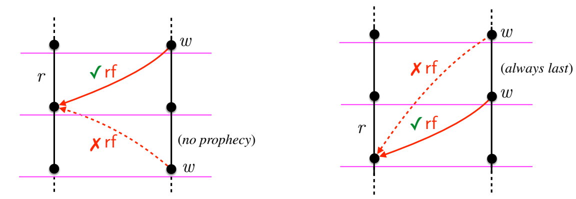
• ***Sequential consistency in*** `cat`. Fig. 11 gives a definition of Sequential Consistency (SC) in `cat`. The intuition is that if e_1 `po`

```
acyclic po | rf | co | fr as sc
```

Figure 11: SC in `cat`

e_2 then event e_1 should appear on a cut before that of e_2 (since instructions are executed in program order), if w `rf` r 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 2010, 2012, Th. 3) shows that



(a) No prophecy beyond cuts (b) Read from last write

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*** H_{com}. 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*** H_{com} ***are allowed by*** S_{com} is done contrapositively *i.e.*, $\neg S_{com} \Rightarrow \neg H_{com}$. By $\neg S_{com}$ in (1), we get $\exists i, j$. $[rf\langle \texttt{F2}^i_4,\ \texttt{0},\ \mathsf{false}\rangle \vee rf\langle \texttt{F2}^i_4,\ \texttt{17},\ \mathsf{false}\rangle \vee rf\langle \texttt{T}^i_5,\ \texttt{11},\ \texttt{1}\rangle] \wedge [rf\langle \texttt{F1}^j_{13},\ \texttt{0},\ \mathsf{false}\rangle \vee rf\langle \texttt{F1}^j_{13},\ \texttt{8},\ \mathsf{false}\rangle \vee rf\langle \texttt{T}^j_{14},\ \texttt{2},\ \texttt{2}\rangle]]$ which we put in disjunctive normal form and give the cases illustrated in Fig. 10, thus proving $\neg H_{com}$.

```
irreflexive fr; po; fr; po
irreflexive fr; po
irreflexive co; po; fr; po
irreflexive rf; po; rf; po
```

and there are no prophecies on cuts.

Figure 13: A possible specification H_{com} of Peterson algorithm

2.3.4 Consistency proof $M \Rightarrow H_{com}$

Proving that all the behaviours allowed by $M = M_i,\ i \in [1, n]$ are allowed by H_{com} is done by reductio ad absurdum in the semantic domain of the consistency specification language (*e.g.* candidate executions for `cat`). Suppose an execution of Peterson that is forbidden by H_{com} yet allowed by M. By definition of H_{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 `cat` speak, SC is modelled as given in Fig. 11. Now, all 4 sequences required to be irreflexive by H_{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 labelised 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_{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 labelised fences as shown in Fig. 14.

```
P0              | P1              ;
f[br] {1} {2} | f[br] {10} {11};
f[br] {2} {4} | f[br] {11} {13};
```

(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 rfe;fence;rfe;fence
irreflexive co;fence;fre;fence
```

Figure 14: Labelised fences for Peterson

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

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

```
let po-loc = po & loc
acyclic po-loc | rf | co | fr as scpv
let ppo = po \ (W*R)
acyclic ppo | rfe | co | fr as tso
```

Figure 15: TSO in `cat`

write event w and the read event r refer to the same variable (as required by `scpv`, see the intersection with the relation `loc`). Else, it is not required on different variables as shown by `tso` (using the relation `ppo` (for *preserved program order*) as the program order po relieved from (see the setminus operator \) the write-read pairs (`W*R`)).

Thus certain executions forbidden by our specification H_{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 `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 labelised fence between 2/4 and 11/13 can be implemented by `mfence` 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.

```
P0                   | P1                   ;
f[cut] (* CS1 *) | f[cut] (* CS2 *);
```

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

```
let cut = (tag2events('cut) * tag2events('cut)) & ext
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 WCM 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_{com} shown to be implied by a program-specific programmer-specified consistency specification H_{com} (*e.g.* in `cat`) itself implied by an architectural consistency specification M (*e.g.* (Shasha and Snir 1988; Alglave 2010; Alglave et al. 2016)). These constraints S_{com} hence H_{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 reshuffles *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 1969) 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 program 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; Batty 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 Demsky 2013; Vafeiadis and Narayan 2013; Turon et al. 2014; Lahav and Vafeiadis 2015; Tassarotti et al. 2015; Doko and Vafeiadis 2016; Lahav et al. 2016) ((Turon et al. 2014) "also considers isolation / ownership transfer properties"), TSO/PSO/RMO in (Burckhardt and Musuvathi 2008; Atig et al. 2010; Wehrman and Berdine 2011; Sieczkowski 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. 2016; Najafzadeh et al. 2016), the "relaxed" memory model in (Burckhardt et al. 2006, 2007), the RMC memory model in (Crary and Sullivan 2015), *etc.*).

We don't have any of the above restrictions since the WCM is a parameter of our proof method and defines the communication relation rf 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*** $\mathtt{P} = \{\mathtt{P}_{\mathsf{start}}\}[\![\mathtt{P}_0 \| \ldots \| \mathtt{P}_{n-1}]\!]$ on shared variables $\mathtt{x} \in \mathsf{loc}[\![\mathtt{P}]\!]$ contain:

- a prelude $\mathtt{P}_{\mathsf{start}}$ 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 `F1` and `F2`, and the value 0 to `T`.
- processes $\mathtt{P}_0 \ldots \mathtt{P}_{n-1}$ in parallel; each process:
  - has an identifier $p \in \mathbb{P}\mathbb{i}[\![\mathtt{P}]\!] \triangleq [0, n[$; in the case of Peterson we have used `P0` for the first process (on the left) and `P1` for the second process (on the right);
  - has local registers (*e.g.* `R0, R1`); 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:`R`));
  - is a sequence of instructions.

Instructions can be:

- *register instructions* `mov R1` *operation*, where the *operation* has the shape *op* `R2` *r-value*:
  - the operator *op* is arithmetic (*e.g.* `add`, `sub`, `mult`) or boolean (*e.g.* `eq`, `neq`, `gt`, `ge`);
  - `R1` and `R2` are local registers;
  - *r-value* is either a local register or a constant;
- *read instructions* `r[`*ts*`] R x` initiate the reading of the value of the shared variable `x` and write it into the local register `R`;
- *write instructions* `w[`*ts*`] x` e initiate the writing of the value of the register expression e into the shared variable `x`;
- *branch instructions* `b[`*ts*`]` *operation* l_t branch to label l_t if the *operation* has value `true` and go on in sequence otherwise;
- *fence instructions* `f[`*ts*`]` $\left[\{l_1^0 \ldots l_1^m\}\ \{l_2^0 \ldots l_2^q\}\right]$. The optional sets of labels $\{l_1^0 \ldots l_1^m\}$ and $\{l_2^0 \ldots l_2^q\}$ indicate that the fence `f[]` only 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 $l_1^0, \ldots,$ or l_1^m and $l_2^0, \ldots, l_2^q$, is to be defined in a `cat` specification;
- *read-modify-write instructions* (RMW) `rmw[`*ts*`] R (`*reg-instrs*`) x` reads shared variable `x` into local register `R`, the register instructions *reg-instrs* are executed, and `R` is written back to `x`. Any semantics requirement on RMWs, such as the fact that there can be no intervening write to `x` between the read and the write of the RMW, has to be ensured by a `cat` specification.

Instructions can be labelled (*i.e.*, be preceded by a control label ℓ) to be referred to in branches or fences for example. Labels are unique; if not we make them different by affixing the process identifier like so: p:l. $\mathsf{instr}[\![\mathtt{P}]\!]_p\ell$ is the instruction at label $\ell \in \mathbb{L}(p)$ of process p of program `P`. Moreover, instructions can bear tags *ts* (to model for example C++ release and acquire annotations).

4.2 Anarchic Semantics

4.2.1 Executions

The anarchic semantics of a parallel program is a set of executions; an execution has the form $\xi = \varsigma \times \pi \times \mathsf{rf} \in \varXi$, where ς is the *computation*, π is the *cut sequence*, and rf is the *communication* part. An example is given in Fig. 2.

Communications are relations rf, which gather *read-from* pairs $\mathit{rf}[w, r]$ linking a (possibly initial) write event w and a read event r relative to the same shared variable `x` with the same value. Communications are *anarchic*: we place no restriction on which write a read can read from; restrictions can be made in a WCM specification however.

Computations have the form $\varsigma = \tau_{\mathsf{start}} \times \prod_{p \in \mathbb{P}\mathbb{i}} \tau_p$, where τ_{start} is an execution *trace* of the prelude process, and τ_p are execution traces of the processes $p \in \mathbb{P}\mathbb{i}$. A finite (resp. infinite) non-empty trace $\tau_p, p \in \mathbb{P}\mathbb{i} \cup \{\mathsf{start}\}$ is a finite (resp. infinite) sequence

$$\tau_p \quad = \quad \langle \xrightarrow{\overline{\tau_p}_k} \underline{\tau_p}_k \mid k \in [0, 1+m[\rangle \quad \in \quad \mathfrak{Tr}$$

of computation steps $\xrightarrow{\overline{\tau_p}_k} \underline{\tau_p}_k$ (with $\overline{\tau_p}_k$ an *event* and $\underline{\tau_p}_k$ the next *state*—see below for the definitions of event and state) such that $\overline{\tau_p}_0 = \epsilon_{\mathsf{start}}$ is the start event (not represented in Fig. 2), $|\tau_p| \triangleq m \in \mathbb{N}_*$ for finite traces and $|\tau_p| \triangleq m = \omega = 1 + \omega$ for infinite traces where ω is the first infinite limit ordinal so that $[0, 1+\omega[\;=\; [0, \omega[\;=\; \mathbb{N}$. Abstracting away the cut sequence π, we get a true parallelism semantics since there is a notion of local time in each trace τ_p, $p \in \mathbb{P}\mathbb{i} \cup \{\mathsf{start}\}$ of an execution $\xi = \tau_{\mathsf{start}} \times \prod_{p \in \mathbb{P}\mathbb{i}} \tau_p \times \mathsf{rf}$ but no global time, since it is impossible to state that an event of a process happens before or after an event of another process or when communications in rf do happen.

Events indicate several things:

- their *nature*, *e.g.* read (`r`), write (`w`), branch (`b`), fence (`f`), *etc.*;
- the *identifier* p of the process that they come from;
- the *control label* ℓ of the instruction that they come from;
- the *instruction* that they come from—which gives the shared variables and local registers affected by the event, if any, *e.g.* `x` and `R` in the case of a read `r[`*ts*`] R x`;
- their *stamp* $\theta \in \mathfrak{T}(p)$; they ensure that events in a trace are unique. In our examples, stamps gather the control label and iteration counters of all surrounding loops, but this is not mandatory: all we need is for events to be uniquely stamped. Different processes have non-comparable stamps. Stamps are totally ordered per process by $\lhd_p$ (which is irreflexive and transitive, while events on different processes are different and incomparable). The successor function succ_p is *s.t.* $\theta \lhd_p \mathsf{succ}_p(\theta)$ (but not necessarily the immediate successor); inf_p is a minimal stamp for process p. We consider executions up to the isomorphic order-preserving renaming $\cong$ of stamps;
- their *value* $v \in \mathcal{D}$, whether ground or symbolic (for writes. as in symbolic execution). To identify the ground or symbolic values that are communicated in invariants, we use pythia variables $\mathfrak{P}(p) \triangleq \{\mathtt{x}_\theta \mid \mathtt{x} \in \mathbb{X} \wedge \theta \in \mathfrak{T}(p)\}$ (note that the uniqueness of stamps on traces ensures the uniqueness of pythia variables for reads). More precisely, traditional methods such as Lamport's

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 $\mathtt{x}_\theta$ the value of shared variable x read at local time θ.

The events $\overline{\tau_p}$ on a trace τ_p of process p are as follows (*e.g.* Fig. 10):

- register events: $\mathfrak{a}(\langle p,\ \ell,\ \mathtt{mov}\ \mathtt{R}_1\ \textit{operation},\ \theta\rangle, v)$;
- read events: $\mathfrak{r}(\langle p,\ \ell,\ \mathtt{r[}\textit{ts}\mathtt{]}\ \mathtt{R}_1\ \mathtt{x},\ \theta\rangle, \mathtt{x}_\theta))$;
- write events: $\mathfrak{w}(\langle p,\ \ell,\ \mathtt{w[}\textit{ts}\mathtt{]}\ \mathtt{x}\ \textit{r-value},\ \theta\rangle, v)$;
- branch events are of two kinds:
  - $\mathfrak{t}(\langle p,\ \ell,\ \mathtt{b[}\textit{ts}\mathtt{]}\ \textit{operation}\ l_t,\ \theta\rangle)$ for the true branch;
  - $\mathfrak{t}(\langle p,\ \ell,\ \mathtt{b[}\textit{ts}\mathtt{]}\ \neg\textit{operation}\ l_t,\ \theta\rangle)$ for the false branch;
- fence events: $\mathfrak{m}(\langle p,\ \ell,\ \mathtt{f[}\textit{ts}\mathtt{]}\ \left[\{l_1^0 \ldots l_1^m\}\ \{l_2^0 \ldots l_2^q\}\right],\ \theta\rangle)$;
- RMW events are of two kinds:
  - begin event: $\mathfrak{m}(\langle p,\ \ell,\ \mathtt{beginrmw[}\textit{ts}\mathtt{]}\ \mathtt{x},\ \theta\rangle)$;
  - end event: $\mathfrak{m}(\langle p,\ \ell,\ \mathtt{endrmw[}\textit{ts}\mathtt{]}\ \mathtt{x},\ \theta\rangle)$

States $\sigma = \mathfrak{s}\langle \ell,\ \theta,\ \rho,\ \nu\rangle$ of a process p mention:

- ℓ, the current control label of process p (we have $\mathsf{done}[\![\mathtt{P}]\!](p)\ell$ which is true if and only if ℓ is the last label of process p which is reached when process p does terminate);
- θ is the stamp of the state in process p;
- ρ is an environment mapping the local registers R of process p to their ground or symbolic value $\rho(\mathtt{R})$;
- ν is a valuation mapping the pythia variables $\mathtt{x}_\theta \in \mathfrak{P}(p)$ of a process p to their ground or symbolic value $\nu(\mathtt{x}_\theta)$. This is a partial map since the pythia variables (*i.e.*, the domain $\mathsf{dom}(\nu)$ of the valuation ν) 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 $\varsigma = \tau_{\mathsf{start}} \times \prod_{p\in\mathbb{P}\mathbb{i}} \tau_p$ is a tuple $\prod_{p\in\mathbb{P}\mathbb{i}}\langle \overline{\tau_p}_k,\ \underline{\tau_p}_k\rangle$ of pairs of events and states of trace τ_p, $p \in \mathbb{P}\mathbb{i}$. 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 $\xi = \varsigma \times \pi \times \mathsf{rf} \in \varXi$.

Conditions over computations $\varsigma = \tau_{\mathsf{start}} \times \prod_{p\in\mathbb{P}\mathbb{i}} \tau_p$ are as follows:

- *Start*: traces τ must all start with a unique fake start event $\epsilon_{\mathsf{start}}$:
  $\overline{\tau}_0 = \epsilon_{\mathsf{start}} \wedge \forall i \in\ ]0, 1+|\tau|[\ .\ \overline{\tau}_i \neq \epsilon_{\mathsf{start}}$. $\qquad \mathsf{Wf}_2(\xi)$
- *Uniqueness*: the stamps of events must be unique:
  $\forall p\ q \in \mathbb{P}\mathbb{i} \cup \{\mathsf{start}\}\ .\ \forall i \in [0, 1+|\tau_p|[\ .\ \forall j \in [0, 1+|\tau_q|[\ .$
  $(p = q \Rightarrow i \neq j) \Rightarrow \mathsf{stamp}(\overline{\tau_p}_i) \neq \mathsf{stamp}(\overline{\tau_q}_j)$. $\qquad \mathsf{Wf}_3(\xi)$
  It immediately follows that events of a trace are unique, and the pythia variable $\mathtt{x}_\theta$ in any read $\mathfrak{r}(\langle p,\ \ell,\ \mathtt{r}\ \mathtt{:=}\ \mathtt{x},\ \theta\rangle, \mathtt{x}_\theta)$ is unique.
- *Initialisation*: all shared variables x are initialised once and only once to a value $v_{\mathtt{x}}$ in the prelude (or to $v_{\mathtt{x}} = 0$ by default).
  $\exists \tau_0\ \tau_1\ .\ \tau_{\mathsf{start}} = \tau_0\ (\mathfrak{w}(\langle \mathsf{start},\ \ell_{\mathsf{start}},\ \mathtt{x}\ \mathtt{:=}\ e,\ \theta\rangle, v_{\mathtt{x}}))\ \tau_1\ \wedge$
  $\forall \tau_0\ \tau_1\ \tau_2\ .\ (\tau_{\mathsf{start}} = \tau_0 \xrightarrow{\epsilon} \sigma\tau_1\ \mathfrak{w}(\langle \mathsf{start},\ \ell,\ \mathtt{x}\ \mathtt{:=}\ e,\ \theta\rangle, v_{\mathtt{x}})\ \tau_2)$
  $\Rightarrow (\epsilon \in \mathfrak{W}(\mathsf{start}, \mathtt{y}) \wedge \mathtt{y} \neq \mathtt{x})$. $\qquad \mathsf{Wf}_4(\xi)$
- *Maximality*: a finite trace τ_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:
  $\exists\ \ell\ \theta\ \rho\ \nu\ .\ \underline{\tau_p}_{|\tau|} = \mathfrak{s}\langle \ell,\ \theta,\ \rho,\ \nu\rangle \wedge \mathsf{done}[\![\mathtt{P}]\!](p)\ell \qquad \mathsf{Wf}_5(\xi)$
  i.e., the control state of the last state of the trace is at the end of the process, as indicated by $\mathsf{done}[\![\mathtt{P}]\!](p)\ell$.

Conditions over the cut sequence π of a computation $\varsigma = \tau_{\mathsf{start}} \times \prod_{p\in\mathbb{P}\mathbb{i}} \tau_p$ are as follows:

- *Start*: The initial cut is $\quad \pi_0 = \prod_{p\in\mathbb{P}\mathbb{i}} \langle \overline{\tau_p}_0,\ \underline{\tau_p}_0\rangle \qquad \mathsf{Wf}_6(\xi)$
- *Step*: But for the final cut, if any, the next cut follows from a computation step. If $\pi_i = \prod_{p\in\mathbb{P}\mathbb{i}} \langle \overline{\tau_p}_{k_{i,p}},\ \underline{\tau_p}_{k_{i,p}}\rangle \wedge \exists q \in \mathbb{P}\mathbb{i}\ .$ $k_{i,q} < |\tau_q|$ then $\exists q \in \mathbb{P}\mathbb{i}$ such that $k_{i,q}+1 \leqslant |\tau_q|$ and
  $$\pi_{i+1} = \prod_{p\in\mathbb{P}\mathbb{i}\setminus\{q\}} \langle \overline{\tau_p}_{k_{i,p}},\ \underline{\tau_p}_{k_{i,p}}\rangle \times \langle \overline{\tau_q}_{k_{i,q}+1},\ \underline{\tau_q}_{k_{i,q}+1}\rangle \qquad \mathsf{Wf}_7(\xi)$$

Conditions over the communications rf are as follows:

- *Satisfaction*: a read event has at least one corresponding communication in rf: $\qquad \mathsf{Wf}_8(\xi)$
  $\forall i \in\ ]0, 1+|\tau_p|[\ .\ \forall r\ .\ (\overline{\tau_p}_i = r) \Rightarrow (\exists w\ .\ \mathit{rf}[w, r] \in \mathsf{rf})$.
- *Singleness*: a read event in the trace τ_p must have at most one corresponding communication in rf: $\qquad \mathsf{Wf}_9(\xi)$
  $\forall r\ w\ w'\ .\ (\mathit{rf}[w, r] \in \mathsf{rf} \wedge \mathit{rf}[w', r] \in \mathsf{rf}) \Rightarrow (w = w')$.
  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:
  $(\mathit{rf}[\mathfrak{w}(\langle p,\ \ell,\ \mathtt{x}\ \mathtt{:=}\ \mathtt{r},\ \theta\rangle, v), \mathfrak{r}(\langle p',\ \ell',\ \mathtt{r}'\ \mathtt{:=}\ \mathtt{x}',\ \theta'\rangle, \mathtt{x}'_\theta)] \in \mathsf{rf})$
  $\Rightarrow (\mathtt{x} = \mathtt{x}')$. $\qquad \mathsf{Wf}_{10}(\xi)$
- *Inception*: no communication is possible without the occurrence of both the read and (maybe initial) write it involves: $\qquad \mathsf{Wf}_{11}(\xi)$
  $\forall r\ w.(\mathit{rf}[w, r] \in \mathsf{rf}) \Rightarrow (\exists p \in \mathbb{P}\mathbb{i}, q \in \mathbb{P}\mathbb{i} \cup \{\mathsf{start}\}.$
  $\exists j \in [0, 1+|\tau_p|[\ .\ \exists k \in [0, 1+|\tau_q|[\ .\ \overline{\tau_p}_j = r \wedge \overline{\tau_q}_k = w)$.
  Note that this does not prevent a read to read from a future write.

Language-dependent conditions for LISA are as follows:

- *Start*: the initial state of a trace τ_p should be of the form:
  $\underline{\tau_p}_0 = \mathfrak{s}\langle l_p^0,\ \mathsf{inf}_p,\ \boldsymbol{\lambda}\,\mathtt{R}\bullet 0,\ \emptyset\rangle \qquad \mathsf{Wf}_{12}(\xi)$
  where l_p^0 is the entry label of process p and inf_p is a minimal stamp of p.
- *Next state*: if at point k of a trace τ_p of process p of an execution $\xi = \tau_{\mathsf{start}} \times \prod_{p\in\mathbb{P}\mathbb{i}} \tau_p \times \pi \times \mathsf{rf}$ the computation is in state $\underline{\tau_p}_{k-1} = \mathfrak{s}\langle \ell,\ \theta,\ \rho,\ \nu\rangle$ then:
  - the next event must be generated by the instruction $\mathit{instr} \triangleq \mathsf{instr}[\![\mathtt{P}]\!]_p\ell$ at label ℓ of process p
  - the next event has the form $\overline{\tau_p}_k = \mathfrak{e}\langle\langle p,\ \ell,\ \mathit{instr},\ \theta\rangle,\ \mathtt{x}_\theta,\ v\rangle$
  - the next state $\underline{\tau_p}_k = \mathfrak{s}\langle \kappa',\ \theta',\ \rho',\ \nu'\rangle$ has $\kappa' = \ell'$ which is the label after the instruction *instr*
  - the stamp $\theta' = \mathsf{succ}_p(\theta)$ is larger, and
  - the value v as well as the new environment ρ' and valuation ν' are computed as a function of the previous environment ρ, the valuation ν, and the execution ξ.

Formally: $\forall k \in\ ]0, 1+|\tau_p|[\ .\ \forall \kappa'\ \rho\ \rho'\ \nu\ \nu'\ \theta\ \theta'$.
$(\underline{\tau_p}_{k-1} = \mathfrak{s}\langle \ell,\ \theta,\ \rho,\ \nu\rangle \wedge \overline{\tau_p}_k = \mathfrak{e}\langle\langle p,\ \ell,\ \mathit{instr},\ \theta\rangle,\ \mathtt{x}_\theta,\ v\rangle\ \wedge$
$\underline{\tau_p}_k = \mathfrak{s}\langle \kappa',\ \theta',\ \rho',\ \nu'\rangle) \Rightarrow (\kappa' = \ell' \wedge \theta' = \mathsf{succ}_p(\theta)\ \wedge$
$v = \boldsymbol{v}(\rho) \wedge \rho' = \boldsymbol{\rho}(v, \rho) \wedge \boldsymbol{\nu}(v, \rho, \nu, \xi, \nu'))$.

We give the form of the next event $\overline{\tau_p}_k$ for each LISA instruction:

- *Fence* ($\mathit{instr} = \ell : \mathtt{f[}\textit{ts}\mathtt{]}\ \left[\{l_1^0 \ldots l_1^m\}\ \{l_2^0 \ldots l_2^q\}\right]; \ell' : \ldots$):
  $\overline{\tau_p}_k = \mathfrak{m}(\langle p,\ \ell,\ \mathtt{f[}\textit{ts}\mathtt{]}\ \left[\{l_1^0 \ldots l_1^m\}\ \{l_2^0 \ldots l_2^q\}\right],\ \theta\rangle)$
  $(\rho' = \rho \wedge \nu' = \nu)$. $\qquad \mathsf{Wf}_{13}(\xi)$
- *Register instruction* ($\mathit{instr} = \ell : \mathtt{mov}\ \mathtt{R}_1\ \textit{operation}; \ell' : \ldots$):
  $\overline{\tau_p}_k = \mathfrak{a}(\langle p,\ \ell,\ \mathtt{mov}\ \mathtt{R}_1\ \textit{operation},\ \theta\rangle, v) \qquad \mathsf{Wf}_{14}(\xi)$
  $(v = \mathit{E}[\![\mathit{operation}]\!](\rho, \nu) \wedge \rho' = \rho[\mathtt{R}_1 := v] \wedge \nu' = \nu)$.
  where $\mathit{E}[\![e]\!](\rho, \nu)$ is the evaluation of the expression e in the environment ρ and valuation ν.
- *Write* ($\mathit{instr} = \ell : \mathtt{w[}\textit{ts}\mathtt{]}\ \mathtt{x}\ \textit{r-value}; \ell' : \ldots$):

$\overline{\tau_{p}}_k = \mathfrak{w}(\langle p,\ \ell,\ \texttt{w[}ts\texttt{]}\ \texttt{x}\ \textit{r-value},\ \theta\rangle, v)$ $\qquad \mathsf{Wf}_{15}(\xi)$
$(v = E[\![\textit{r-value}]\!](\rho,\nu) \wedge \rho = \rho' \wedge \nu' = \nu)$.

- *Read* (*instr* = ℓ : $\texttt{r[}ts\texttt{]}\ \texttt{R}_1\ \texttt{x};\ell' : \ldots$):
  $\overline{\tau_{p}}_k = \mathfrak{r}(\langle p,\ \ell,\ \texttt{r[}ts\texttt{]}\ \texttt{R}_1\ \texttt{x},\ \theta\rangle, \texttt{x}_\theta)$ $\qquad \mathsf{Wf}_{16}(\xi)$
  $(\rho' = \rho[\texttt{R}_1 := \texttt{x}_\theta] \wedge \exists q \in \mathbb{P}\mathfrak{i} \cup \{\mathsf{start}\}\ .\ \exists! j \in [1, 1 + |\tau_q|[\ .$
  $\exists \ell'', \theta'', v\ .\ (\overline{\tau_{q}}_j = \mathfrak{w}(\langle q,\ \ell'',\ \texttt{w[}ts\texttt{]}\ \texttt{x}\ \textit{r-value},\ \theta''\rangle, v) \wedge$
  $\mathsf{rf}[\overline{\tau_{q}}_j, \overline{\tau_{p}}_k] \in \mathsf{rf} \wedge \nu' = \nu[\texttt{x}_\theta := v]))$.
- *RMW* (*instr* = $\texttt{rmw[}ts\texttt{]}\ \texttt{r}\ \texttt{(}\textit{reg-instrs}\texttt{)}\ \texttt{x}$): for the begin (*instr* = $\texttt{beginrmw[}ts\texttt{]}\ \texttt{x}$) and end event (*instr* = $\texttt{endrmw[}ts\texttt{]}\ \texttt{x}$):
  $\overline{\tau_{p}}_k = \mathfrak{m}(\langle p,\ \ell,\ \textit{instr},\ \theta\rangle)$ $\qquad \mathsf{Wf}_{17}(\xi)$
  $(\rho' = \rho \wedge \nu' = \nu)$.
- *Test* (*instr* = ℓ : $\texttt{b[}ts\texttt{]}$ *operation* $l_t; \ell' : \ldots$):
  - on the true branch:
    $\overline{\tau_{p}}_k = \mathfrak{t}(\langle p,\ \ell,\ \texttt{b[}ts\texttt{]}\ \textit{operation}\ l_t,\ \theta\rangle)$ $\qquad \mathsf{Wf}_{18t}(\xi)$
    $(\mathsf{sat}(E[\![\textit{operation}]\!](\rho,\nu) \neq 0) \wedge \kappa' = l_t \wedge \rho' = \rho \wedge \nu' = \nu)$
  - on the false branch:
    $\overline{\tau_{p}}_k = \mathfrak{t}(\langle p,\ \ell,\ \texttt{b[}ts\texttt{]}\ \neg\textit{operation}\ l_t,\ \theta\rangle)$ $\qquad \mathsf{Wf}_{18f}(\xi)$
    $(\mathsf{sat}(E[\![\textit{operation}]\!](\rho,\nu) = 0) \wedge \kappa' = \ell' \wedge \rho' = \rho \wedge \nu' = \nu)$

4.2.3 Anarchic semantics

The anarchic semantics of a program P is

$$S^{\mathsf{a}}[\![\mathtt{P}]\!] \quad \triangleq \quad \{\xi \in \varXi \mid \mathsf{Wf}_2(\xi) \wedge \ldots \wedge \mathsf{Wf}_{18}(\xi)\}\ .$$

Theorem 1 *In an execution $\xi = \varsigma \times \pi \times \mathsf{rf} \in \varXi$, the communication* rf *uniquely determines the computation ς.*

4.2.4 Consistency specification of a semantics

The semantics $S[\![H_{com}]\!] \in \varXi \to \{\texttt{allowed}, \texttt{forbidden}\}$ of a consistency specification H_{com} checks whether an execution $\xi \in \varXi$ is allowed or forbidden by H_{com}. Defining the *consistency abstraction* $\alpha_{\mathsf{ana}}[\![H_{com}]\!](S) \triangleq \{\xi \in S \mid S[\![H_{com}]\!]\xi = \texttt{allowed}\}$, we have the Galois connection $\langle \wp(\varXi), \subseteq\rangle \xleftarrow[\alpha_{\mathsf{ana}}[\![H_{com}]\!]]{\gamma_{\mathsf{ana}}[\![H_{com}]\!]} \langle \wp(\varXi), \subseteq\rangle$. The analytic semantics of a program P for the consistency specification H_{com} is then $S[\![\mathtt{P}]\!] \triangleq \alpha_{\mathsf{ana}}[\![H_{com}]\!](S^{\mathsf{a}}[\![\mathtt{P}]\!])$.

Example 1 (cat specification) The candidate execution abstraction $\alpha_\varXi(\xi)$ abstracts the execution $\xi = \varsigma \times \pi \times \mathsf{rf} \in \varXi$ into a candidate execution $\alpha_\varXi(\xi) = \langle e, po, rf, iw\rangle$ where e is the set of events in ς (partitionned into fence, read, write, … events), po is the program order (transitively relating successive events on a trace of each process), rf = rf is the set of communications, and iw is the set of initial write events. Then we define $\alpha_\varXi(S) \triangleq \{\langle \xi,\ \alpha_\varXi(\xi)\rangle \mid \xi \in S\}$ and $\alpha_{\text{🐱}}[\![H_{com}]\!](C) \triangleq \{\xi \mid \exists \Gamma\ .\ \langle \xi,\ \varXi\rangle \in C \wedge \langle \texttt{allowed}, \Gamma\rangle \in \text{🐱}[\![H_{com}]\!](\varXi)\}$ where the consistence 🐱$[\![H_{com}]\!](\varXi)$ of a candidate execution $\varXi$ for a cat consistency model H_{com} is defined in (Alglave et al. 2016) and returns communication relations Γ specifying communication constraints on communication events (*e.g.* containing `co` of `with co from AllCo` in H_{com}). The consistency abstraction for a cat specification H_{com} is then $\alpha_{\mathsf{ana}}[\![H_{com}]\!] \triangleq \alpha_{\text{🐱}}[\![H_{com}]\!] \circ \alpha_\varXi$. □

5. Invariance Abstraction

The semantics $S[\![\mathtt{P}]\!]$ of a program P is a set of executions $\xi \in \varXi$ so belongs to $\wp(\varXi)$. Representing properties by the set of elements which have this property, ***semantic properties*** $\mathcal{P}$ are elements of $\mathcal{P} \in \wp(\wp(\varXi))$. So P has semantic property $\mathcal{P}$ means $S[\![\mathtt{P}]\!] \in \mathcal{P}$, equivalently $\{S[\![\mathtt{P}]\!]\} \subseteq \mathcal{P}$ where $\{S[\![\mathtt{P}]\!]\}$ is the strongest semantic property (called *collecting semantics*) and $\subseteq$ is implication.

The join abstraction $\alpha_\cup(\mathcal{P}) = \cup\mathcal{P}$ such that $\langle \wp(\wp(\varXi)), \subseteq\rangle \xleftarrow[\alpha_\cup]{\gamma_\cup} \langle \wp(\varXi), \subseteq\rangle$ yields ***execution properties*** $P \in \wp(\varXi)$. So P has execution property P means $S[\![\mathtt{P}]\!] \in \gamma_\cup(P)$ that is $\{S[\![\mathtt{P}]\!]\} \subseteq \gamma_\cup(P)$ equivalently $\alpha_\cup(\{S[\![\mathtt{P}]\!]\}) \subseteq P$ *i.e.*, $S[\![\mathtt{P}]\!] \subseteq P$. The strongest execution property of P is $S[\![\mathtt{P}]\!]$.

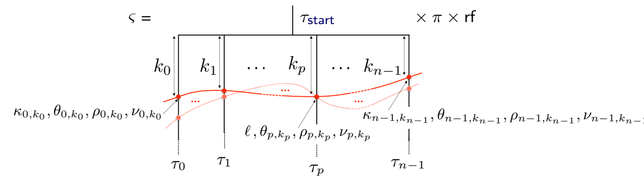


Figure 17: Invariance abstraction of an execution

The ***invariance abstraction*** $\alpha_{\mathsf{inv}}(P)$, $P \in \wp(\varXi)$ on Fig. 17 collects states on all cuts of all traces at each control point of each process.

$$\alpha_{\mathsf{inv}}(P) \quad \triangleq \quad \prod_{p \in \mathbb{P}\mathfrak{i}} \prod_{\ell \in \mathbb{L}(p)} \bigcup_{\xi \in P} \alpha_{\mathsf{inv}}(\xi)_p(\ell) \qquad (19)$$

$\alpha_{\mathsf{inv}}(\tau_{\mathsf{start}} \times \prod_{p=0}^{n-1} \tau_p \times \pi \times \mathsf{rf})_p(\ell) \triangleq \{\langle \kappa_{0,k_0}, \theta_{0,k_0}, \rho_{0,k_0}, \nu_{0,k_0},$ $\ldots, \nu_{p-1,k_{p-1}}, \theta_{p,k_p}, \rho_{p,k_p}, \nu_{p,k_p}, \kappa_{p+1,k_{p+1}}, \ldots, \kappa_{n-1,k_{n-1}},$ $\theta_{n-1,k_{n-1}}, \rho_{n-1,k_{n-1}}, \nu_{n-1,k_{n-1}}, \mathsf{rf}\rangle \mid \forall q \in [0, n[\ .\ \forall k_q < |\tau_q|\ .$ $\underline{\tau_q}_{k_q} = \mathfrak{s}\langle \kappa_{q,k_q},\ \theta_{q,k_q},\ \rho_{q,k_q},\ \nu_{q,k_q}\rangle \wedge \kappa_{p,k_p} = \ell\}$.

An ***invariance property*** $I \in \mathfrak{I}$, in particular the strongest invariant $\alpha_{\mathsf{inv}}(S[\![\mathtt{P}]\!]) \in \mathfrak{I}$, attaches a ***local invariance property*** $I_p(\ell)$ at each program point ℓ of each process p, 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 ℓ of process p. We have $\langle \wp(\varXi), \subseteq\rangle \xleftarrow[\alpha_{\mathsf{inv}}]{\gamma_{\mathsf{inv}}} \langle \mathfrak{I}, \dot\subseteq\rangle$ so P has invariance property $S_{inv} \in \mathfrak{I}$ means $S[\![\mathtt{P}]\!] \in \gamma_\cup(\gamma_{\mathsf{inv}}(S_{inv}))$ *i.e.*, $S[\![\mathtt{P}]\!] \subseteq \gamma_{\mathsf{inv}}(S_{inv})$ that is $\alpha_{\mathsf{inv}}(S[\![\mathtt{P}]\!]) \dot\subseteq S_{inv}$.

In practice local invariance properties are often expressed as logical formulæ $S_{ind\,p}(\ell)$ attached to program points $\ell \in \mathbb{L}(p)$ of each process $p \in \mathbb{P}\mathfrak{i}$ which logical interpretation is a set-theoretic property in $\mathfrak{I}$. Formally, a logical assertion S_{ind} is a logical formula $S_{ind\,p}(\ell)$ with free variables κ_0, θ_0, ρ_0, ν_0, …, ν_{p-1}, θ_p, ρ_p, ν_p, κ_{p+1} …, κ_{n-1}, θ_{n-1}, ρ_{n-1}, ν_{n-1}, and rf attached to each program point ℓ of each process p of the program (excluding $\kappa_p = \ell$).

The assertions on control are often written $\mathsf{at}_p\{\ell\}$ (or $\mathsf{at}\{\ell\}$ if the label ℓ is unique to process p) to mean that $\kappa_p = \ell$. We write $\mathsf{at}_p\{\ell_1, \ldots, \ell_m\}$ for $\bigvee_{\ell=\ell_1}^{\ell_m} \mathsf{at}_p\{\ell\}$. Moreover the assertions on environments and valuations are expressed using assertions on registers and pythia variables. For example, $\rho \in R$ is expressed by the logical formula $\forall \rho \in R\ .\ \bigwedge_{\mathtt{r} \in \mathsf{dom}(\rho)} \mathtt{r} = \rho(\mathtt{r})$, or any equivalent logical formula. The initial values of shared variables is determined by the prelude (0 by default) so $S_{ind\,p,\ell}$ states assertional properties. For relational invariance (Cousot and Cousot 1982) the initial value of shared variables is set to an initial pythia variable.

6. Calculational Design of the Proof Method

Given a program P and an invariance specification S_{inv}, the invariance proof method consists in proving that $\alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}[\![H_{com}]\!](S^{\mathsf{a}}[\![\mathtt{P}]\!])) \dot\subseteq S_{inv}$. The design of the invariance proof method by calculus starts as follows ($\Leftarrow$ is soundness and $\Rightarrow$ is completeness):

$\alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}[\![H_{com}]\!](S^{\mathsf{a}}[\![\mathtt{P}]\!])) \dot\subseteq S_{inv}$
$\Leftrightarrow \alpha_{\mathsf{inv}}(\{\xi \in S^{\mathsf{a}}[\![\mathtt{P}]\!] \mid S[\![H_{com}]\!]\xi = \texttt{allowed}\}) \dot\subseteq S_{inv}$ (def. $\alpha_{\mathsf{ana}}[\![H_{com}]\!]$)
$\Leftrightarrow \alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathtt{P}]\!] \cap \{\xi \in S^{\mathsf{a}}[\![\mathtt{P}]\!] \mid S[\![H_{com}]\!]\xi = \texttt{allowed}\}) \dot\subseteq S_{inv}$ (def. $\cap$)
$\Leftrightarrow \alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathtt{P}]\!]) \cap \alpha_{\mathsf{inv}}(\{\xi \in \varXi \mid S[\![H_{com}]\!]\xi = \texttt{allowed}\}) \dot\subseteq S_{inv}$
(since α_{inv} preserves intersections)
$\Leftrightarrow \alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathtt{P}]\!]) \dot\cap \alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}[\![H_{com}]\!](S^{\mathsf{a}}[\![\mathtt{P}]\!])) \dot\subseteq S_{inv}$ (def. $\alpha_{\mathsf{ana}}[\![H_{com}]\!]$)
$\Leftrightarrow \exists S_{com}\ .\ \alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathtt{P}]\!]) \dot\cap S_{com} \dot\subseteq S_{inv} \wedge \alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}[\![H_{com}]\!](S^{\mathsf{a}}[\![\mathtt{P}]\!])) \dot\subseteq S_{com}$
(($\Leftarrow$) For soundness, we have $\alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathtt{P}]\!]) \dot\cap \alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}[\![H_{com}]\!](S^{\mathsf{a}}[\![\mathtt{P}]\!])) \dot\subseteq \alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathtt{P}]\!]) \dot\cap S_{com} \dot\subseteq S_{inv}$;

(⇒) For completeness, we choose to describe exactly the communications that is $\mathsf{S}_{com} = \alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}[\![\mathsf{H}_{com}]\!](\mathsf{S}^{\mathsf{a}}[\![\mathrm{P}]\!]))$.∫

$$\Leftrightarrow \exists \mathsf{S}_{com} \,.\, (\mathsf{S}_{com} \Rightarrow \mathsf{S}_{inv}) \wedge (\mathsf{H}_{com} \Rightarrow \mathsf{S}_{com})$$

by defining the conditional invariance proof $\mathsf{S}_{com} \Rightarrow \mathsf{S}_{inv}$ to be $\alpha_{\mathsf{inv}}(\mathsf{S}^{\mathsf{a}}[\![\mathrm{P}]\!]) \dot{\cap} \mathsf{S}_{com} \dot{\subseteq} \mathsf{S}_{inv}$ and the inclusion proof $\mathsf{H}_{com} \Rightarrow \mathsf{S}_{com}$ to be $\alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}[\![\mathsf{H}_{com}]\!](\mathsf{S}^{\mathsf{a}}[\![\mathrm{P}]\!])) \dot{\subseteq} \mathsf{S}_{com}$.

This calculation justifies the decomposition of the correctness proof in Fig. 4 into an invariance proof and an inclusion proof using an intermediate communication specification S_{com}.

7. Conditional Invariance Verification Conditions

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

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*

$$\mathsf{PRE}^{\ell,\kappa}_{p,r} \triangleq \mathsf{S}_{ind\,p}(\ell)[\kappa_r \leftarrow \kappa, \theta_r, \rho_r, \nu_r, \mathsf{rf}] \wedge \mathsf{S}_{ind\,r}(\kappa)[\kappa_p \leftarrow \ell, \theta_r, \rho_r, \nu_r, \mathsf{rf}]$$

$$\mathsf{PRE}^{\ell}_{p} \triangleq \mathsf{PRE}^{\ell,\ell}_{p,p} = \mathsf{S}_{ind\,p}(\ell)[\theta_p, \rho_p, \nu_p, \mathsf{rf}]$$

(where $\mathsf{A}[x_1, \ldots, x_m]$ stipulates that A has, among others, free variables $x_1, \ldots, x_m$, $\mathsf{A}[x \leftarrow e]$ is the substitution of e for x with renaming of quantified variables to avoid variable capture, and PRE^{ℓ}_{p} does not depend upon κ_p so the substitution $[\kappa_p \leftarrow \ell]$ has no effect.)

• *Postcondition*

$$\mathsf{POST}^{\ell,\kappa'}_{p,r} \triangleq (\mathsf{S}_{com_p} \stackrel{.}{\Longrightarrow} \mathsf{S}_{ind\,p})(\ell)[\kappa_r \leftarrow \kappa', \theta_r \leftarrow \mathsf{succ}_r(\theta_r)]$$

$$\mathsf{POST}^{\ell'}_{p} \triangleq \mathsf{POST}^{\ell',\ell'}_{p,p} = (\mathsf{S}_{com_p} \stackrel{.}{\Longrightarrow} \mathsf{S}_{ind\,p})(\ell')[\theta_p \leftarrow \mathsf{succ}_p(\theta_p)]$$

(where $\mathsf{POST}^{\ell'}_{p}$ does not depend upon κ_p so the substitution $[\kappa_p \leftarrow \ell']$ has no effect.)

• *Communication condition*

$$\mathsf{COM}^{\ell}_{p}[\mathsf{rf}] \triangleq \mathsf{S}_{ind\,p}(\ell)[\mathsf{rf}] \wedge \mathsf{S}_{com_p}(\ell)[\mathsf{rf}]$$

meaning that a read or write instruction at ℓ of process p may execute (since the invariant $\mathsf{S}_{ind\,p}(\ell)[\mathsf{rf}]$ holds) and communicate according to rf (as specified by $\mathsf{S}_{com_p}(\ell)[\mathsf{rf}]$).

7.2 Initialisation Verification Condition

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

• $\mathsf{PRE}^{\ell^0_p}_{p} \prod_{r \in \mathbb{P}\mathbb{i}} [\kappa_r \leftarrow \ell^0_r, \theta_r \leftarrow \mathsf{inf}_r, \rho_r \leftarrow \boldsymbol{\lambda}\,\mathtt{R}\bullet 0, \nu_r \leftarrow \emptyset]$

(where $\mathsf{A} \prod_{i=1}^{m} [x_i \leftarrow e_i]$ is $\mathsf{A}[x_1 \leftarrow e_1] \ldots [x_m \leftarrow e_m]$.)

7.3 Sequential Verification Conditions

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

7.4 Non-interference Verification Conditions

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

• For local side-effect free ***marker instructions*** κ : *instr* κ' where *instr* = `f[`*ts*`]` $\left[\{l^0_1 \ldots l^m_1\}\ \{l^0_2 \ldots l^q_2\}\ \right]$, `w[`*ts*`] x` *r-value*, `beginrmw[`*ts*`] x`, `endrmw[`*ts*`] x`: (marker)

$$\mathsf{PRE}^{\ell,\kappa}_{p,r} \Rightarrow \mathsf{POST}^{\ell,\kappa'}_{p,r}$$

• For the ***register assignment*** κ : `mov R` *operation* κ'; (assign)

$$\mathsf{PRE}^{\ell,\kappa}_{p,r}[\rho_r, \nu_r] \Rightarrow \mathsf{POST}^{\ell,\kappa'}_{p,r}[\rho_r \leftarrow \rho_r[\mathtt{R} := \mathsf{E}[\![operation]\!](\rho_r, \nu_r)]]$$

• For a ***read instruction*** κ : `r[`*ts*`] R x` κ': (read)

$$\mathsf{PRE}^{\ell,\kappa}_{p,r}[\theta_r, \rho_r, \nu_r, \mathsf{rf}] \wedge rf[\mathfrak{w}(\langle q,\ \ell',\ \mathtt{w[}ts\mathtt{]\ x}\ r\text{-}value,\ \theta'\rangle, v), \mathfrak{r}(\langle r,\ \ell,\ \mathtt{r[}ts\mathtt{]\ R\,x},\ \theta_r\rangle, \mathtt{x}_{\theta_r})] \in \mathsf{rf}$$

$$\Rightarrow \mathsf{POST}^{\ell,\kappa'}_{p,r}[\rho_r \leftarrow \rho_r[\mathtt{R} := \mathtt{x}_{\theta_r}], \nu_r \leftarrow \nu_r[\mathtt{x}_{\theta_r} := v]]$$

• For a ***test instruction*** κ : `b[`*ts*`]` *operation* l_t κ': (test)

$$\mathsf{PRE}^{\ell,\kappa}_{p,r}[\rho_r, \nu_r] \wedge \mathsf{sat}(\mathsf{E}[\![operation]\!](\rho_r, \nu_r) \neq 0) \Rightarrow \mathsf{POST}^{\ell,l_t}_{p,r}$$

$$\mathsf{PRE}^{\ell,\kappa}_{p,r}[\rho_r, \nu_r] \wedge \mathsf{sat}(\mathsf{E}[\![operation]\!](\rho_r, \nu_r) = 0) \Rightarrow \mathsf{POST}^{\ell,\kappa'}_{p,r}$$

where 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 "$\{P_i\}S_i\{Q_i\}$, $i \in [1, n]$ *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 ℓ : `r[`*ts*`] R x` ℓ' must read either from an initial or a reachable program write, allowed by the communication hypothesis ($\boldsymbol{\exists}\,\mathsf{P}[X_1, \ldots, X_m]$ means that all free variables in predicate P but $X_1, \ldots, X_m$ are existentially quantified):

$$\mathsf{COM}^{\ell}_{p}[\theta_p, \mathsf{rf}] \wedge \mathsf{rf} \neq \emptyset \Rightarrow \exists\, rf[\mathfrak{w}(\langle q,\ \ell_q,\ \mathtt{w[}ts\mathtt{]\ x}\ r\text{-}value,\ \theta'\rangle, v), \mathfrak{r}(\langle p,\ \ell,\ \mathtt{r[}ts\mathtt{]\ R\,x},\ \theta_p\rangle, \mathtt{x}_{\theta_p})] \in \mathsf{rf}\ . \quad \text{(satisfaction)}$$

$$((q \in \mathbb{P}\mathbb{i} \wedge \boldsymbol{\exists}\,\mathsf{PRE}^{\ell_q}_{q}[\theta_q \leftarrow \theta', \mathsf{rf}]) \vee (q = \mathsf{start} \wedge v = 0))\ .$$

• A read event can read from only one write event.

$$\mathsf{COM}^{\ell}_{p}[\mathsf{rf}] \wedge rf[r, w_1] \in \mathsf{rf} \wedge rf[r, w_2] \in \mathsf{rf} \Rightarrow w_1 = w_2\ . \quad \text{(singleness)}$$

• The values v allowed to be read by the communication hypothesis must originate from reachable program write instructions ℓ : `w[`*ts*`] x` *r-value* ℓ':

$$\forall \mathsf{rf}\ .\ \forall\, rf[\mathfrak{w}(\langle q,\ \ell_q,\ \mathtt{w[}ts\mathtt{]\ x}\ r\text{-}value,\ \theta_p\rangle, v), r] \in \mathsf{rf} \quad \text{(match)}$$

$$\mathsf{COM}^{\ell}_{p}[\theta_q, \rho_q, \nu_q, \mathsf{rf}] \Rightarrow v = \mathsf{E}[\![r\text{-}value]\!](\rho_q, \nu_q)$$

• The inception condition $\mathsf{Wf}_{11}(\xi)$ is not required since non-existent communications can only lead to more imprecise invariants, which is sound. However, it is always possible to take inception into account to get precise invariants for completeness.

The communications taken into account in rf must include all those of the anarchic semantics as restricted by S_{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 $rf\langle \mathtt{x}_{\ell_p},\ \ell_q,\ v\rangle$ for $rf[\mathfrak{w}(\langle q,\ \ell_q,\ \mathtt{w[}ts\mathtt{]\ x}\ r\text{-}value,\ \ell_q\rangle, v), \mathfrak{r}(\langle p,\ \ell_p,\ \mathtt{r[}ts\mathtt{]\ R1\ x},\ \ell_p\rangle, \mathtt{x}_{\ell_p})]$ and define $\Gamma_t \triangleq \{rf\langle \mathtt{x}_1,\ 0,\ 0\rangle, rf\langle \mathtt{x}_1,\ 7,\ 42\rangle\} \times \{rf\langle \mathtt{y}_5,\ 0,\ 0\rangle, rf\langle \mathtt{y}_5,\ 3,\ 42\rangle\}$.

`{ 0: w[] x 0; w[] y 0;` $\{\mathtt{x=0} \wedge \mathtt{y=0}\}$ `}`

`1:` $\{\mathtt{R1=0} \wedge \mathsf{rf} \in \Gamma_t\}$	`5:` $\{\mathtt{R2=0} \wedge \mathsf{rf} \in \Gamma_t\}$
`r[] R1 x` $\{\rightsquigarrow \mathtt{x}_1\}$	`r[] R2 y` $\{\rightsquigarrow \mathtt{y}_5\}$;
`2:` $\{\mathtt{R1=x}_1 \wedge \mathtt{x}_1 \in \{0,42\} \wedge \mathsf{rf} \in \Gamma_t\}$	`6:` $\{\mathtt{R2=y}_5 \wedge \mathtt{y}_5 \in \{0,42\} \wedge \mathsf{rf} \in \Gamma_t\}$
`b[] (R1 =/= 42) 4`	`b[] (R2 =/= 42) 8;`
`3:` $\{\mathtt{R1=x}_1 \wedge \mathtt{x}_1\mathtt{=42} \wedge \mathsf{rf} \in \Gamma_t\}$	`7:` $\{\mathtt{R2=y}_5 \wedge \mathtt{y}_5\mathtt{=42} \wedge \mathsf{rf} \in \Gamma_t\}$
`w[] y R1`	`w[] x R2;`
`4:` $\{\mathtt{R1=x}_1 \wedge \mathtt{x}_1 \in \{0,42\}\}$	`8:` $\{\mathtt{R2=y}_5 \wedge \mathtt{y}_5 \in \{0,42\}\}$

By the communication proof for any $\mathsf{rf} \in \Gamma_t$, communicated values cannot be different (match), 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, $\mathtt{x}_1 \in \{0\}$ is prevented by the read rule, all communicated values are readable. Values must come from writes so $\mathtt{x}_1 \in \{0, 42, 43\}$ at $\mathsf{2}$ is prevented by (match). In case of an unconditional branch `b true 8;` at $\mathsf{6}$, any $rf\langle \mathtt{x}_1,\ 7,\ \_\rangle \in \mathsf{rf}$ is prevented by (satisfaction) *i.e.*, it is not possible to read from a non-reachable write. ☐

8. Calculation of the Invariance Proof $S_{com} \Rightarrow S_{inv}$, Soundness and Completeness by Design

The calculation for the invariance proof $S_{com} \Rightarrow S_{inv}$, formally $\alpha_{\mathsf{inv}}(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket) \mathbin{\dot\cap} S_{com} \mathrel{\dot\subseteq} S_{inv}$, goes on by induction on the length of trace prefixes, which requires the use of the inductive invariant S_{ind}. The basis $\mathsf{Wf}_{12}(\xi)$ yields the initialisation condition. The sequential verification condition for $S_{ind\,p}(\ell)$ is obtained when performing a computation step $\mathsf{Wf}_{13}(\xi), \ldots, \mathsf{Wf}_{18}(\xi)$ 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 $\mathsf{Wf}_{8}(\xi)$ to $\mathsf{Wf}_{11}(\xi)$.

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_{inv} is true for the program anarchic semantics $S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket$ with communications restricted by the specification S_{com}). Reciprocally, the proof method is complete so that if an invariance statement S_{inv} is true for the program anarchic semantics $S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket$ with communications restricted by the specification S_{com} then this can always be proved thanks to a stronger inductive invariant S_{ind} satisfying all verification conditions.

As usual the completeness proof provides no clue on how to choose the inductive invariant S_{ind} since it is based on the choice $S_{ind} = \alpha_{\mathsf{inv}}(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket) \mathbin{\dot\cap} S_{com}$, *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 prove is identical using the interpretation of the logical fomulæ. This is a problem however for the completeness proof since, in general, $S_{ind} = \alpha_{\mathsf{inv}}(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket) \mathbin{\dot\cap} S_{com}$ 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 $\alpha_{\mathsf{inv}}(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket) \mathbin{\dot\cap} S_{com}$ is expressible in the logic (Cook 1978, 1981).

Theorem 2 (Invariance proof $\mathbf{S_{com} \Rightarrow S_{inv}}$) *$S_{com} \Rightarrow S_{inv}$, formally $\alpha_{\mathsf{inv}}(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket) \mathbin{\dot\cap} S_{com} \mathrel{\dot\subseteq} S_{inv}$, if and only if there exists $S_{ind} \mathrel{\dot\subseteq} S_{inv}$ 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_{ind} reduce to (Lamport 1977) proof method for sequential consistency.*

Our invariance proof method for WCM is also an extension of (Owicki and Gries 1976) for sequential consistency since, by the argument given in (Cousot and Cousot 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 $H_{com} \Rightarrow S_{com}$, Soundness and Completeness by Design

The calculation of the inclusion proof $H_{com} \Rightarrow S_{com}$, formally $\alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}\llbracket H_{com} \rrbracket(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket)) \mathrel{\dot\subseteq} S_{com}$, yields the verification conditions for the communication specification S_{com} in Th. 4 below. Define $S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P} = \alpha_{\mathsf{ana}}\llbracket H_{com} \rrbracket(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket) = \{\xi \in S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket \mid S\llbracket H_{com} \rrbracket\xi = \mathtt{allowed}\}$. We have

$$
\begin{aligned}
& \alpha_{\mathsf{inv}}(\alpha_{\mathsf{ana}}\llbracket H_{com} \rrbracket(S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket)) \mathrel{\dot\subseteq} S_{com} \\
\Leftrightarrow\ & \alpha_{\mathsf{inv}}(S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P}) \mathrel{\dot\subseteq} S_{com} && \text{\{def. } S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P}\text{\}} \\
\Leftrightarrow\ & \forall \xi \in S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P}\ .\ \alpha_{\mathsf{inv}}(\{\xi\}) \mathrel{\dot\subseteq} S_{com} && \text{\{}\alpha_{\mathsf{inv}}\text{ preserves } \cup\text{\}} \\
\Leftrightarrow\ & \forall \xi \in S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P}\ .\ \dot{\bigcup_{p=1}^{n}} \dot{\bigcup_{\mathtt{L} \in \mathtt{P}_p}} \{\alpha_{\mathsf{inv}}(\xi')_p(\mathtt{L}) \mid \xi' \in \{\xi\}\} \mathrel{\dot\subseteq} S_{com} \\
& && \text{\{def. (19) of } \alpha_{\mathsf{inv}}\text{\}} \\
\Leftrightarrow\ & \forall (\tau_{\mathsf{start}} \times \prod_{p=0}^{n-1} \tau_p \times \pi \times \mathsf{rf}) \in S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P}\ .\ \forall p \in [1, n]\ .\ \forall \mathtt{L} \in \mathtt{P}_p\ . \\
& \quad \alpha_{\mathsf{inv}}(\tau_{\mathsf{start}} \times \prod_{p=0}^{n-1} \tau_p \times \pi \times \mathsf{rf})_p(\mathtt{L}) \subseteq S_{com\,p}(\mathtt{L})
\end{aligned}
$$

{def. $\in$, $\dot\bigcup$, $\dot\subseteq$, and $S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P}$ so that ξ has the form $\xi = \tau_{\mathsf{start}} \times \prod_{p=0}^{n-1} \tau_p \times \pi \times \mathsf{rf}$. By def. (19) of α_{inv} and $\subseteq$, we get}

$$
\begin{aligned}
\Leftrightarrow\ & \forall (\tau_{\mathsf{start}} \times \prod_{p=0}^{n-1} \tau_p \times \pi \times \mathsf{rf}) \in S^{\mathsf{ana}}\llbracket H_{com} \rrbracket\mathtt{P}\ .\ \forall i \in [1, n]\ .\ \forall \mathtt{L} \in \mathtt{P}_p\ .\ \forall q \in [0, n[\ .\ \forall k_q < |\tau_q|\ . \\
& (\underline{\tau_q}_{k_q} = \mathfrak{s}\langle \kappa_{q,k_q},\ \theta_{q,k_q},\ \rho_{q,k_q},\ \nu_{q,k_q}\rangle \wedge \kappa_{p,k_p} = \mathtt{L}) \Rightarrow \\
& \langle \kappa_{0,k_0}, \theta_{0,k_0}, \rho_{0,k_0}, \nu_{0,k_0}, \ldots, \nu_{p-1,k_{p-1}}, \theta_{p,k_p}, \rho_{p,k_p}, \nu_{p,k_p}, \\
& \kappa_{p+1,k_{p+1}}, \ldots, \kappa_{n-1,k_{n-1}}, \theta_{n-1,k_{n-1}}, \rho_{n-1,k_{n-1}}, \nu_{n-1,k_{n-1}}, \mathsf{rf}\rangle \\
& \in S_{com\,i}(\mathtt{L})
\end{aligned} \tag{20}
$$

i.e., any cut of any anarchic execution history allowed by the consistency specification H_{com} must satisfy all local invariants S_{com} along the cut.

So we have proved

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

Observe that the completeness proof of Th. 4 assumes that $H_{com} \Rightarrow S_{com}$. If the consistency specification language is not expressive enough there might be no way to express a strong enough consistency specification H_{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_{com}. This means that the design of the program P must ensure that S_{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 $H_{com} \Rightarrow S_{com}$ using an anarchic invariant *i.e.*, an invariant of the anarchic semantics.

10. Anarchic Invariant

An anarchic invariant S^{a}_{inv} of the anarchic semantics $S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket$ 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.* `[r[] R1 x; R1=R1+1; w[] R1 x]` with no feasible execution on $\mathbb{Z}$).

The anarchic semantics $S^{\mathsf{a}}\llbracket \mathtt{P} \rrbracket$ considers all possible write events $\overline{W}\llbracket \mathtt{P} \rrbracket(\theta_q, \mathtt{x})$, $q \in \mathbb{P}\mathbb{i} \cup \{\mathsf{start}\}$, $\theta_q \in \mathfrak{T}(q)$

$$
\begin{aligned}
\overline{W}\llbracket \mathtt{P} \rrbracket(\theta_q, \mathtt{x}) \triangleq \{ & \mathfrak{w}(\langle q,\ \ell_q,\ \mathtt{w[}ts\mathtt{]}\ \mathtt{x}\ r\text{-}value,\ \theta_q\rangle, v_{\theta_q}) \mid \ell_q \in \mathbb{L}(q) \\
& \wedge \mathsf{instr}\llbracket \mathtt{P} \rrbracket^{\ell_q}_q = \mathtt{w[}ts\mathtt{]}\ \mathtt{x}\ r\text{-}value \wedge v_{\theta_q} \in \mathcal{D}\}
\end{aligned}
$$

and all possible read events $\overline{R}[\![\mathrm{P}]\!](\theta_p, \mathtt{x})$, $p \in \mathbb{P}\mathrm{i}$, $\theta_p \in \mathfrak{T}(p)$.

$$\overline{R}[\![\mathrm{P}]\!](\theta_p, \mathtt{x}) \triangleq \{\mathfrak{r}(\langle p,\ \ell_p,\ \mathtt{r}\,[ts]\ \mathtt{R}\,\mathtt{x},\ \theta_p\rangle, \mathtt{x}_{\theta_p}) \mid \ell_p \in \mathbb{L}(p) \wedge \mathsf{instr}[\![\mathrm{P}]\!]_p^{\ell_p} = \mathtt{r}\,[ts]\ \mathtt{R}\,\mathtt{x}\}$$

The anarchic communications $\mathsf{rf} \in \overline{\Gamma}[\![\mathrm{P}]\!]$ are between matching write and read events for any shared variable $\mathtt{x} \in \mathsf{loc}[\![\mathrm{P}]\!]$.

$$\overline{\Gamma}[\![\mathrm{P}]\!] \triangleq \Big\{\{rf[w_{\theta_q,\mathtt{x}}, r_{\theta_p,\mathtt{x}}] \mid \mathtt{x} \in \mathsf{loc}[\![\mathrm{P}]\!] \wedge q \in \mathbb{P}\mathrm{i} \cup \{\mathsf{start}\} \wedge \theta_q \in \mathfrak{T}(q) \wedge p \in \mathbb{P}\mathrm{i} \wedge \theta_p \in \mathfrak{T}(p)\} \Bigm| \forall \mathtt{x}, q, \theta_q, p, \theta_p \ .\ w_{\theta_q,\mathtt{x}} \in \overline{W}[\![\mathrm{P}]\!](\theta_q, \mathtt{x}) \wedge r_{\theta_p,\mathtt{x}} \in \overline{R}[\![\mathrm{P}]\!](\theta_p, \mathtt{x})\Big\}$$

Define an *anarchic invariant* to be

$$S^{\mathsf{a}}_{inv} \triangleq \bigcup\{S^{\mathsf{a}}_{ind}[\mathsf{rf}] \mid \mathsf{rf} \in \overline{\Gamma}[\![\mathrm{P}]\!] \wedge S^{\mathsf{a}}_{ind}[\mathsf{rf}] \text{ is inductive for } \mathrm{P} \text{ and } S_{com} = \mathsf{true}\}.$$

Theorem 5 (Anarchic invariant) *S_{inv} is an invariant of the anarchic semantics $S^{\mathsf{a}}[\![\mathrm{P}]\!]$ (i.e., $\alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathrm{P}]\!]) \mathrel{\dot\subseteq} S_{inv}$) if and only if there exists an anarchic invariant S^{a}_{inv} such that $S^{\mathsf{a}}_{inv} \mathrel{\dot\subseteq} S_{inv}$.*

S^{a}_{inv} can be obtained by considering the values v_{θ_q} 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_{θ_q}.

Example 2 (Thin air 2) Consider the following program mono-process P. The possible anarchic communications are $\overline{\Gamma}[\![\mathrm{P}]\!] \triangleq \{\{rf\langle \mathtt{x}_1,\ 0,\ v_0\rangle \mid v_0 \in \mathcal{D}\}\} \cup \{\{rf\langle \mathtt{x}_1,\ 3,\ v_3\rangle \mid v_3 \in \mathcal{D}\}\} \cup \{\{rf\langle \mathtt{x}_1,\ 4,\ v_4\rangle\} \mid v_4 \in \mathcal{D}\}\}$.
Define

$$\mathsf{com}_0 \triangleq \mathtt{x}_1 = v_0 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 0,\ v_0\rangle\}$$
$$\mathsf{com}_3 \triangleq \mathtt{x}_1 = v_3 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 3,\ v_3\rangle\}$$
$$\mathsf{com}_4 \triangleq \mathtt{x}_1 = v_4 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 4,\ v_4\rangle\}$$

```
0:{ x = 0 } [
1:r[] R1 x; {⇝x1}
2:if R1=42 then
3:   w[] x 41;
4:   w[] x R1;
5:fi; ]
6:
```

where v_0, v_3, and v_4 are fresh symbolic variables. Consider the following invariant (including the terms overlined in red).

0: `{ x = 0 } [`
1: $\{\mathtt{R1} = 0 \wedge (\mathsf{com}_0 \vee \cancel{\mathsf{com}_3 \vee} \mathsf{com}_4)\}$
   `r[] R1 x;` $\{\rightsquigarrow \mathtt{x}_1\}$
2: $\{\mathtt{R1} = \mathtt{x}_1 \wedge (\mathsf{com}_0 \vee \cancel{\mathsf{com}_3 \vee} \mathsf{com}_4)\}$
   `if R1=42 then`
3:   $\{\ \mathtt{R1} = \mathtt{x}_1 = 42 \wedge (\cancel{\mathsf{com}_0 \vee \mathsf{com}_3 \vee} \mathsf{com}_4)\}\ \}$
     `w[] x 41;`
4:   $\{\ \mathtt{R1} = \mathtt{x}_1 = 42 \wedge (\cancel{\mathsf{com}_0 \vee \mathsf{com}_3 \vee} \mathsf{com}_4)\}\ \}$
     `w[] x R1;`
5:   $\{\ \mathtt{R1} = \mathtt{x}_1 = 42 \wedge (\cancel{\mathsf{com}_0 \vee \mathsf{com}_3 \vee} \mathsf{com}_4)\}\ \}$
   `fi;`
6: $\{\mathtt{R1} = \mathtt{x}_1 \wedge (\mathsf{com}_0 \vee \cancel{\mathsf{com}_3 \vee} \mathsf{com}_4)\}$ `]`

By the (match) rule at 0 $v_0 = 0$, at 3 $v_3 = 41$, and at 4 $v_4 = \mathtt{x}_1 = 42$. It follows that com_0 and com_3 are false at 3, 4 and 5. By the (satisfaction) rule for the read 1, com_3 where $v_3 = 41$ must hold at 3 and it doesn't so com_3 does not hold at 1. Then, by the (read) rule com_3 is false at 1 hence, by the (test) rule, com_3 is false 6. We get the anarchic inductive invariant S^{a}_{inv} (excluding the infeasible communications of the terms overlined in red which are false according to the program semantics).

0: `{ x = 0 } [`
1: $\{\mathtt{R1} = 0 \wedge ((\mathtt{x}_1 = 0 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 0,\ 0\rangle\}) \vee (\mathtt{x}_1 = 42 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 4,\ 42\rangle\})))\}$
   `r[] R1 x;` $\{\rightsquigarrow \mathtt{x}_1\}$
2: $\{\mathtt{R1} = \mathtt{x}_1 \wedge ((\mathtt{x}_1 = 0 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 0,\ 0\rangle\}) \vee (\mathtt{x}_1 = 42 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 4,\ 42\rangle\})))\}$
   `if R1=42 then`
3:   $\{\ \mathtt{R1} = 42 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 4,\ 42\rangle\}\ \}$
     `w[] x 41;`
4:   $\{\ \mathtt{R1} = 42 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 4,\ 42\rangle\}\ \}$
     `w[] x R1;`
5:   $\{\ \mathtt{R1} = 42 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 4,\ 42\rangle\}\ \}$
   `fi;`
6: $\{\ (\mathtt{R1} = 0 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 0,\ 0\rangle\}) \vee (\mathtt{R1} = 42 \wedge \mathsf{rf} = \{rf\langle \mathtt{x}_1,\ 4,\ 42\rangle\})\ \}$ `]` □

11. Candidate Executions for `cat` Specifications

The proof of $H_{com} \Rightarrow S_{com}$ by Th. 4 requires extracting executions from the semantics $S^{\mathsf{ana}}[\![H_{com}]\!]\mathrm{P} = \alpha_{\mathsf{ana}}[\![H_{com}]\!](S^{\mathsf{a}}[\![\mathrm{P}]\!]) = \alpha_{☺}[\![H_{com}]\!] \circ \alpha_{\Xi}(S^{\mathsf{a}}[\![\mathrm{P}]\!])$ *i.e.*, candidate executions $\alpha_{\Xi}(S^{\mathsf{a}}[\![\mathrm{P}]\!])$ for `cat` specifications. Th. 6 shows that it is sound to extract a superset of the anarchic candidate executions $\alpha_{\Xi}(S^{\mathsf{a}}[\![\mathrm{P}]\!])$ directly from an anarchic invariant S^{a}_{inv} (excluding no 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^{a}_{inv} (*i.e.*, satisfying the invariance verification conditions of Sect. 7 with $S_{com} = \mathsf{true}$). Moreover, by the completeness result, the extraction can be exact by choosing a precise enough anarchic invariant.

Theorem 6 (Candidate executions out of an anarchic invariant) *The over-approximation of the anarchic program candidate executions $\{\alpha_{\Xi}(\xi) \mid \xi \in S^{\mathsf{a}}[\![\mathrm{P}]\!]\}$ by extraction $\alpha^{\mathsf{a}}_{\Xi}(S^{\mathsf{a}}_{inv})$ from an anarchic invariant S^{a}_{inv} is sound ($\subseteq$) and complete ($\supseteq$).*

where the set of candidate executions defined by an assertion A at point ℓ of process p for a program P is

$$\alpha^{\mathsf{a}}_{\Xi}(A) \triangleq \bigcup\{\alpha^{\mathsf{a}}_{\Xi\,p,l}(A) \mid p \in \mathbb{P}\mathrm{i} \wedge \ell \in \mathbb{L}(p)\}$$
$$\alpha^{\mathsf{a}}_{\Xi\,p,l}(A) \triangleq \{\alpha^{\mathsf{a}}_{\Xi}(A, \mathsf{rf}') \mid \mathsf{rf}' \in C_{p,\ell}(A)\}$$

and the possible communication relations are

$$C_{p,\ell}(A) \triangleq \{\mathsf{rf}' \mid \exists \kappa_0, \theta_0, \rho_0, \nu_0, \ldots, \nu_{p-1}, \theta_p, \rho_p, \nu_p, \kappa_{p+1}, \ldots, \kappa_{n-1}, \theta_{n-1}, \rho_{n-1}, \nu_{n-1}\ .\ A_{p,\ell}[\mathsf{rf} \leftarrow \mathsf{rf}']\}$$

$$\alpha^{\mathsf{a}}_{\Xi}(A, \mathsf{rf}') \triangleq \langle e(A, \mathsf{rf}'),\ po(A, \mathsf{rf}'),\ \mathsf{rf}',\ iw\rangle$$

$$e(A, \mathsf{rf}') \triangleq \{\mathsf{N}(\ell)(p, \ell, \mathsf{instr}[\![\mathrm{P}]\!]^{\ell}_{p}, \theta[, \mathtt{x}_\theta][, v]) \mid p \in \mathbb{P}\mathrm{i} \wedge \ell \in \mathbb{L}(p) \wedge \exists \kappa_0, \theta, \rho_0, \nu_0, \ldots, \nu_{p-1}, \rho_p, \nu_p, \kappa_{p+1}, \ldots, \kappa_{n-1}, \theta_{n-1}, \rho_{n-1}, \nu_{n-1}\ .\ A_{p,\ell}[\theta_p \leftarrow \theta, \mathsf{rf} \leftarrow \mathsf{rf}']\} \cup iw$$

$\mathsf{N}(\ell) \triangleq$ $\mathfrak{w}$, $\mathfrak{r}$, $\mathfrak{t}$, or $\mathfrak{m}$ whether the LISA instruction $\mathsf{instr}[\![\mathrm{P}]\!]^{\ell}_{p}$ at ℓ of p is a write to x, read of x, test, fence, or begin/end of read-modify-write (register assignments are ignored in `cat` specifications)

$$iw \triangleq \{\mathfrak{w}(\langle \mathsf{start},\ \ell_{\mathsf{start}},\ \mathtt{x} := \mathtt{e},\ \theta\rangle, v_\theta) \mid \mathtt{x} \in \mathsf{loc}[\![\mathrm{P}]\!]\}$$

$$po(A, \mathsf{rf}') \triangleq (iw \times (e(A, \mathsf{rf}') \setminus iw)) \cup \{\langle \mathsf{N}(\ell)(p, \ell, instr, \theta[, \mathtt{x}]), \mathsf{N}(\ell')(p, \ell', instr', \theta'[, \mathtt{y}])\rangle \in e(A, \mathsf{rf}')^2 \mid p \in \mathbb{P}\mathrm{i} \wedge \theta \lhd_p \theta'\}\ .$$

Considering local process invariants, we get Cor. 7 ensuring that all possible anarchic communications allowed by the progran semantics are taken into account when proving S_{com}.

Corollary 7 (Candidate executions out of local anarchic invariants) *For all $p \in \mathbb{P}\mathrm{i}$ and $\ell \in \mathbb{L}(p)$, the extraction $\alpha^{\mathsf{a}}_{\Xi\,p,l}(S^{\mathsf{a}}_{inv_p}(\ell))$ of the program candidate executions from an anarchic invariant S^{a}_{inv} (i.e., $\alpha_{\mathsf{inv}}(S^{\mathsf{a}}[\![\mathrm{P}]\!]) \subseteq S^{\mathsf{a}}_{inv}$) is sound (i.e., $\{\alpha_{\Xi}(\xi) \mid \xi \in \gamma_{\mathsf{inv}\,p,\ell}(S^{\mathsf{a}}_{inv_p}(\ell))\} \subseteq \alpha^{\mathsf{a}}_{\Xi\,p,l}(S^{\mathsf{a}}_{inv_p}(\ell))$) and complete.*

12. Inclusion Proof Revisited for `cat` Specs

To prove $H_{com} \Rightarrow S_{com}$ (without reasoning directly on the anarchic semantics $S^{\mathsf{a}}[\![\mathrm{P}]\!]$), 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^{a}_{inv} by def. $C(S^{\mathsf{a}}_{inv})$, 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, hence essentially the communication relation, is consistent with the WCM defined by H_{com}. If this is the case, it should be accepted by the communication specification invariant S_{com}. It follows that S_{com} does not forget any feasible (allowed by the program semantics/logic) and consistent (allowed by H_{com}) communication relation, hence by Th. 1 execution.

Theorem 8 (Inclusion proof for `cat`) *$H_{com} \Rightarrow S_{com}$ (formally $\alpha_{\mathsf{inv}}(\alpha_{☺}[\![H_{com}]\!] \circ \alpha_{\Xi}(S^{\mathsf{a}}[\![\mathrm{P}]\!])) \mathrel{\dot\subseteq} S_{com}$) if and only if there exists an anarchic invariant S^{a}_{inv} satisfying the following* (inclusion) *verification condition:*

$$\forall p \in \mathbb{P}\mathfrak{i} \, . \, \forall \ell \in \mathbb{L}(p) \, . \, \forall \mathsf{rf}' \in C_{p,\ell}(S^{\mathsf{a}}_{inv_p}(\ell)) \, . \qquad \text{(inclusion)}$$
$$\big(\text{🐱}[\![H_{com}]\!](\alpha^{\mathsf{a}}_{\Xi}(S^{\mathsf{a}}_{inv_p}(\ell), \mathsf{rf}')) \wedge S^{\mathsf{a}}_{inv_p}(\ell)[\mathsf{rf} \leftarrow \mathsf{rf}']\big)$$
$$\Rightarrow S_{com_p}(\ell)[\mathsf{rf} \leftarrow \mathsf{rf}'] \, .$$

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

`let` r `=` $R(\langle e,\ po,\ rf,\ iw\rangle)$

`acyclic` | `irreflexive` | `empty` | `not empty` r

where the relation $r \in \wp(e \times e)$ is a function R of e, po, rf, and iw, as defined by the `cat` language semantics 🐱$[\![H_{com}]\!]$ (Alglave et al. 2016). We have `acyclic`(r) if and only if `irreflexive`(r^+) so we only have to handle irreflexivity and emptyness.

The proof $\neg S_{com_p}(\ell)[\mathsf{rf}] \ \Rightarrow\ \neg$🐱$[\![H_{com}]\!](\alpha^{\mathsf{a}}_{\Xi}(S^{\mathsf{a}}_{inv}[\mathsf{rf}]))$ is by contraposition, *i.e.*, any communication rejected by S_{com} is also rejected by H_{com}. Assuming $\neg S_{com_p}(\ell)[\mathsf{rf}]$, the check $\neg$🐱$[\![H_{com}]\!](\langle e,\ po,\ rf,\ iw\rangle)$ considers each of these reflexivity or emptyness 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 $S_{inv} \triangleq \neg(\mathtt{at}\{8\} \wedge \mathtt{at}\{28\})$ plus non-starvation (CSs are entered infinitely often).

Anarchic communications of PostgreSQL. The anarchic communications $\overline{\Gamma}$ are given in Fig. 21. We write $\mathfrak{rf}\langle x_\theta,\ \langle \ell\mathtt{:},\ \theta',\ v\rangle\rangle$ to state that the read into pythia variable x_θ was from an write event marked θ' of value v generated by the action at process label ℓ. The markers θ/θ' are the vectors of iteration counters of the loops enclosing the read/write instruction. We write $\mathrm{R}vp^\theta$ for the possible read-froms of variable `latch`p ($v = $L) or variable `flag`$p$ ($v = $F) in process `P`$p$, $p \in \{0,1\}$ for unique stamp θ (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 $\Gamma \in \overline{\Gamma}$, $\Gamma = \{\mathrm{rl0}^i_{j_i}, \mathrm{rf0}^i, \mathrm{rl1}^\ell_{m_\ell}, \mathrm{rf1}^\ell \mid i \in \mathbb{N} \wedge j_i \in [0, k_i] \wedge \ell \in \mathbb{N} \wedge j \in [0, n_\ell]\}$ encodes a particular read-from relation `rf` specifying that for the i^{th} iteration in the external loop 1–12 and the $j_i{}^{\text{th}}$ iteration in the internal loop 2–4 of process `P0`, `3: r[] Rl0 latch0` $\{\leadsto L0^i_{j_i}\}$ will read as specified by $\mathrm{rl0}^i_{j_i} \in \mathrm{RL0}^i_{j_i}$ while `6: r[] Rf0 flag0` $\{\leadsto F0^i\}$ will read as specified by $\mathrm{rf0}^i \in \mathrm{RF0}^i$.

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

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

It follows that $(S_{com}(\Gamma, \Gamma) \wedge S_{ind}(\Gamma, \Gamma)) \ \Rightarrow\ S_{inv}(\Gamma, \Gamma)$ so $S_{com}(\Gamma, \overline{\Gamma}) \Rightarrow S_{inv}(\Gamma, \overline{\Gamma})$ (since $S_{com}(\Gamma, \overline{\Gamma}) \Rightarrow S_{ind}(\Gamma, \overline{\Gamma})$), proving mutual exclusion under the S_{com} communication hypothesis.

The proof that $S_{com}(\Gamma, \Gamma)$ is also necessary is done by providing counter-examples. For example, a candidate execution counter-example to S_{com_1} 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 ▶).

Consistency specification H_{com} for mutual exclusion. The consistency specification H_{com} is obtained by excluding all executions $S_{com_i}(\Gamma, \overline{\Gamma})$, $i \in [1, 4]$ for the anarchic communications $\overline{\Gamma}$.

For $S_{com_1}(\Gamma, \overline{\Gamma})$, Fig. 20 corresponds to a incorrect scenario where process `P1` reads from the initialisation, enters its critical section, `P0` reads the `latch0` and `flag0` to be later set by process `P1` and so also enters its critical section. This is excluded on architectures with no prophecy beyond cuts (see Fig. 12a where no read before a cut can read from a write after the cut, hence when the write is not yet executed). Another correct scenario would have process `P1` reads from the initialisation, enters and exits its critical section, writes `latch0` and `flag0`, later read by process `P0` which in turn enters its critical section. Both scenarios abstract to the same candidate execution (since cuts are abstracted away) so the incorrect scenario can hardly be excluded in `cat` without referring to cuts (by adding synchronisation markers in case of prophecy beyond cuts, see Fig. 16).

Non-starvation. We must prove that there exists no single execution ξ such that either from the initialisation or from a later local time in this execution, one process (or both) never enter their critical section (under the non-blocking and communication satisfaction $\mathsf{Wf}_{\mathbf{8}}(\xi)$ fairness hypotheses), which yields six cases. We assume there is one such execution ξ, and derive a contradiction. Let Γ_{rf} be the encoding of the communication relation `rf` of this execution ξ. Γ_{rf} uniquely determines `rf`, hence by Th. 1, uniquely determine the execution ξ. The inductive invariant $S_{ind}(\Gamma, \{\Gamma_{\mathsf{rf}}\})$ of Fig. 21 holds for this single execution ξ. By completeness in Th. 2, the strongest such inductive invariant $S_{ind}(\Gamma, \{\Gamma_{\mathsf{rf}}\}) \triangleq \alpha_{\mathsf{inv}}(\{\xi\})$ exists and 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 contradiction is that this is not the case. (This reasoning is not possible with (Owicki and Gries 1976; Lamport 1977) since the inductive invariant holds for communications wired in the interleaved semantics hence in the verification conditions. If every execution ultimately never reaches some critical section, the conclusion is that some execution ultimately reaches the critical section. This does not mean non-starvation, which is to show that every execution ultimately reaches the critical section. The difficulty with (Owicki and Gries 1976; Lamport 1977) is to prove liveness using sets of states (Pnueli et al. 2005; Podelski and Rybalchenko 2005). Here the reasoning is on only one erroneous execution, which is shown impossible. Moreover, we don't need a ranking function on a well-founded set (Pnueli et al. 2005) since there are only two processes with a fixed scheduling). Let us consider two of these cases.

• Assume process `P1` never enters its critical section on the erroneous execution ξ. Since process `P1` never enters its critical section on this execution ξ, the strongest inductive invariant $S_{inv}(\Gamma, \{\Gamma_{\mathsf{rf}}\})$ is false $\mathtt{at}\{28\}$ hence false $\mathtt{at}\{28, 29, 30, 31\}$. So $\neg\mathrm{r1Rf1}^\ell[\Gamma_{\mathsf{rf}}]$ hence $\mathrm{r0Rf1}^\ell[\Gamma_{\mathsf{rf}}]$ holds. By the (satisfaction) verification condition the read $\mathtt{at}\{26\}$ must be from a reachable write of value 0 to `flag1`. The only possible one $\mathtt{at}\{28\}$ is not reachable so this strongest inductive invariant $S_{inv}(\Gamma, \{\Gamma_{\mathsf{rf}}\})$ cannot satisfy the verification conditions, the desired contradiction.

• Assume process `P0` never enters its critical section on the erroneous execution ξ. The strongest inductive invariant $S_{inv}(\Gamma, \{\Gamma_{\mathsf{rf}}\})$ is false $\mathtt{at}\{8\}$ (hence $\mathtt{at}\{8, 9, 10, 11\}$).

So by the invariant $\mathtt{at}\{7\}$ must have $\neg\mathrm{r1Rf0}^i[\Gamma_{\mathsf{rf}}]$ and so $\mathrm{r1Rl0}^i_{k_i}[\Gamma_{\mathsf{rf}}] \wedge \mathrm{r0Rf0}^i[\Gamma_{\mathsf{rf}}]$.

By def. of $\mathrm{r1Rl0}^i_{j_i}[\Gamma_{\mathsf{rf}}] \triangleq \exists \ell_{30} \in \mathbb{N} \, . \, \mathfrak{rf}\langle L0^i_{j_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \Gamma_{\mathsf{rf}} \wedge L0^i_{j_i} = 1$, the read of 1 $\mathtt{at}\{3\}$ is from $\mathtt{at}\{30\}$, which cannot be unreachable by the previous case. By def. of $\mathrm{r1Rl1}^\ell_{m_\ell}[\Gamma_{\mathsf{rf}}] \triangleq$

[1] `www.postgresql.org/message-id/24241.1312739269@sss.pgh.pa.us`

$(\mathfrak{rf}\langle L1^{\ell}_{m_\ell},\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma}_{\mathsf{rf}} \wedge L1^{\ell}_{m_\ell} = 1) \vee (\exists i_{10} \in \mathbb{N}\ .\ \mathfrak{rf}\langle L1^{\ell}_{m_\ell},$ $\langle \mathtt{10:},\ i_{10},\ 1\rangle\rangle \in \mathit{\Gamma}_{\mathsf{rf}} \wedge L1^{\ell}_{m_\ell} = 1)$ the read of 1 $\mathtt{at}\{23\}$ is from $\mathtt{at}\{0\}$ since $\mathtt{at}\{10\}$ is unreachable.

By def. of $\mathrm{r0Rf0}^{i}[\mathit{\Gamma}_{\mathsf{rf}}]$, the read of 0 $\mathtt{at}\{6\}$ has only two possibilities (*i.e.*, a read from 0 or from 8).

1. Either $\mathfrak{rf}\langle F0^{i},\ \langle \mathtt{0:},\ \_,\ 0\rangle\rangle \in \mathit{\Gamma}_{\mathsf{rf}} \wedge F0^{i} = 0$, which is the scenario of Fig. 20. This is prevented using LISA fences with the following cat specification.

```
with co from AllCo                                let flw = fencerel(Flw)
let fencerel(S) = ((po&(_*S));po)&fromto(S) let fcs = deps | flw
let Fdep = F & tag2events('fdep)                  let fr = rf^-1;co
let deps = fencerel(Fdep) & (R * _)               let fre = fr & ext
let Flw = F & tag2events('flw)                    acyclic fre;fcs;rfe;fcs
```

2. Or $\exists i_8 \in \mathbb{N}\ .\ \mathfrak{rf}\langle F0^{i},\ \langle \mathtt{8:},\ i_8,\ 0\rangle\rangle \in \mathit{\Gamma} \wedge F0^{i} = 0$, but the read is from an unreachable write (in the critical section of P0 assumed to be never entered on execution ξ), which is impossible by the (satisfaction) verification condition.

This LISA fence can be implemented for SC (`fences = po`), TSO (`fences = po`, this incorrect communication for PostgreSQL is natively forbidden in TSO without using `mfence`), for PowerPC (`deps = addr | data flw = lwsync`), and ARM (`flw = dmb | dsb`). Other cases are handled similarly, the final result cumulates all LISA fences.

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 labelised 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 values. 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 refencing.

Other proof methods can be derived by applying the proof method transformations of (Cousot and Cousot 1982) *e.g.* backward reasonings or by reductio ad absurdum. Alternative, more modular, proof methods can de 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

M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

J. Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7, 2010.

J. Alglave. A formal hierarchy of weak memory models. Form. Methods Syst. Des. (2012) 41:178210.

J. Alglave and P. Cousot. Syntax and analytic semantics of LISA. *CoRR*, abs/1608.06583, 2016. URL `http://arxiv.org/abs/1608.06583`.

J. Alglave and L. Maranget. herd7. `virginia.cs.ucl.ac.uk/herd`, 31 Aug. 2015.

J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. *ESOP 2013*, LNCS 7792, 512–532. Springer, 2013.

J. Alglave, P. Cousot, and L. Maranget. Syntax and semantics of the weak consistency model specification language `cat`. *CoRR*, abs/1608.07531, 2016. URL `http://arxiv.org/abs/1608.07531`.

M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. *ACM POPL 2010*, 7–18.

M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. *CAV 2011*, LNCS 6806, 99–115. Springer, 2011.

R. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Asp. Comput.*, 2(3):247–272, 1990.

G. Barthe, C. Kunz, and J. L. Sacchini. Certified reasoning in memory hierarchies. *APLAS 2008*, LNCS 5356, 75–90. Springer, 2008.

M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In Giacobazzi and Cousot (2013), 235–248.

R. Bodík and R. Majumdar, editors. *ACM Proceedings of POPL 2016*.

R. Bornat, J. Alglave, and M. J. Parkinson. New lace and arsenic: adventures in weak memory with a program logic. *CoRR*, abs/1512.01416, 2015. URL `http://arxiv.org/abs/1512.01416`.

G. Boudol, G. Petri, and B. P. Serpette. Relaxed operational semantics of concurrent programming languages. *EXPRESS/SOS 2012*, 19–33, 2012.

S. D. Brookes. A denotational approach to weak memory concurrency. *MFPS XXXII*, LNCS. Springer, 2016.

S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. CAV 2008, LNCS 5123, 107–120. Springer, 2008.

S. Burckhardt, R. Alur, and M. M. K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. CAV 2006, LNCS 4144, 489–502. Springer, 2006.

S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. *ACM PLDI 2007*, 12–21.

E. Cohen. Coherent causal memory. *CoRR*, abs/1404.2187, 2014. URL `http://arxiv.org/abs/1404.2187`.

E. Cohen and B. Schirmer. From total store order to sequential consistency: A practical reduction theorem. *ITP 2010*, LNCS 6172, 403–418. Springer, 2010.

J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.

S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

S. A. Cook. Corrigendum: Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 10(3):612, 1981.

P. Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, 303–342, Prentice-Hall, 1981.

P. Cousot and R. Cousot. Reasoning about program invariance proof methods. Res. rep. CRIN-80-P050, Centre de Recherche en Informatique de Nancy (CRIN), Institut National Polytechnique de Lorraine, Nancy, France, July 1980.

P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. In *Tools & Notions for Program Construction: an Advanced Course*, 75–119. CUP, Cambridge, UK, 1982.

P. Cousot, R. Cousot, and R. Giacobazzi. Abstract interpretation of resolution-based semantics. *Theor. Comput. Sci.*, 410(46):4724–4746, 2009.

K. Crary and M. J. Sullivan. A calculus for relaxed memory. In Rajamani and Walker (2015), 623–636.

A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. *VMCAI 2015*, LNCS 8931, 449–466. Springer, 2015.

T. D'Hondt, editor. *ECOOP 2010*, LNCS 6183, 2010. Springer.

T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In D'Hondt (2010), 504–528.

M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In Rajamani and Walker (2015), 233–246.

M. Doko and V. Vafeiadis. A program logic for C11 memory fences. *VMCAI 2016*, LNCS 9583, 413–430. Springer, 2016.

A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In Giacobazzi and Cousot (2013), 129–142.

R. W. Floyd. Assigning meaning to programs. *Proc. Symp. in Applied Math.*, volume 19, 19–32. Amer. Math. Soc., 1967.

R. Giacobazzi and R. Cousot, editors. *ACM Proceedings of POPL 2013*.

A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In Bodík and Majumdar (2016), 371–384.

I. Grief. *Semantics of communicating parallel processes*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, Sept. 1975. URL `https://dspace.mit.edu/handle/1721.1/57710`.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors. *ACM Proceedings of OOPSLA 2013*.

R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. *ICFP 2016*, 256–269.

V. Klebanov. A jmm-faithful non-interference calculus for Java. *FIDJI 2004*, LNCS 3409, 101–111. Springer, 2004.

O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. *ICALP 2015*, LNCS 9135, 311–323. Springer, 2015.

O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In Bodík and Majumdar (2016), 649–662.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), 2012.

A. Miné. Relational thread-modular static value analysis by abstract interpretation. VMCAI 2014, LNCS 8318, 39–58. Springer, 2014.

M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. *TACAS 2007*, LNCS 4424, 568–582. Springer, 2007.

M. O. Myreen, A. C. J. Fox, and M. J. C. Gordon. Hoare logic for ARM machine code. *FSEN 2007*, LNCS 4767, 272–286. Springer, 2007.

M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. The CISE tool: proving weakly-consistent applications correct. *ACM PaPoCEuroSys 2016*, 2:1–2:3.

P. Naur. Proofs of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

B. Norris and B. Demsky. CDSCHECKER: checking concurrent data structures written with C/C++ atomics. In Hosking et al. (2013), 131–150.

P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In D'Hondt (2010), 478–503.

S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

J. Palsberg and M. Abadi, editors. *ACM Proceedings of POPL 2005*.

G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

A. Pnueli, A. Podelski, and A. Rybalchenko. Separating fairness and well-foundedness for the analysis of fair discrete systems. *TACAS, LNCS 3440*, 124–139. Springer, 2005.

A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In Palsberg and Abadi (2005), 132–144.

S. K. Rajamani and D. Walker, editors. *ACM Proceedings of POPL 2015*.

D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

F. Sieczkowski, K. Svendsen, L. Birkedal, and J. Pichon-Pharabod. A separation logic for fictional sequential consistency. *ESOP 2015*, LNCS 9032, 736–761. Springer, 2015.

J. Tassarotti, D. Dreyer, and V. Vafeiadis. Verifying read-copy-update in a logic for weak memory. *ACM PLDI 2015*, 110–120.

A. M. Turing. Checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines, Mathematical Laboratory, Cambridge, UK*, 67–69, 24 June 1949. Reproduced as "An early program proof by Alan Turing", in F.L. Morris and C.B. Jones (Eds), *Annals of the History of Computing*, Vol. 6, Apr. 1984, `http://www.turingarchive.org/browse.php/B/8`.

A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *ACM OOPSLA 2014*, 691–707.

V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In Hosking et al. (2013), 867–884.

I. Wehrman and J. Berdine. A proposal for weak-memory local reasoning. *LOLA workshop*, volume 11, 55–70, 2011.

$$S_{com_1} \triangleq \neg(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29} \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge \mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma}$$

$$S_{com_2} \triangleq \neg(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29}, i_9 \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge \mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{9:},\ i_9,\ 1\rangle\rangle \in \mathit{\Gamma}$$

$$S_{com_3} \triangleq \neg(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29}, i_{10} \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge \mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{10:},\ i_{10},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma}$$

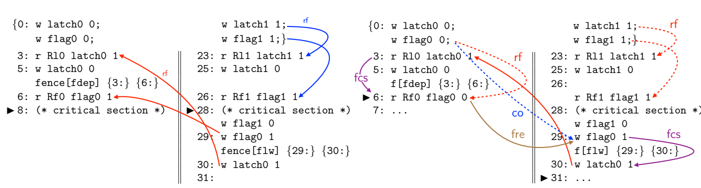
$$S_{com_4} \triangleq \neg(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29}, i_{10}, i_9 \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge \mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{10:},\ i_{10},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge \mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{9:},\ i_9,\ 1\rangle\rangle \in \mathit{\Gamma}$$

Figure 18: Communication specification for PostgreSQL

$$
\begin{array}{ll}
 & (\neg S_{inv}(\mathit{\Gamma}, \Gamma)) \wedge S_{ind}(\mathit{\Gamma}, \Gamma) \\
\triangleq & \mathsf{at}\{8\} \wedge \mathsf{at}\{28\} \wedge S_{ind}(\mathit{\Gamma}, \Gamma) \qquad \wr\text{def. invariance specification } S_{inv}\wr \\
\Rightarrow & \mathsf{at}\{8\} \wedge \mathsf{at}\{28\} \wedge (\exists i, k_i, \ell, n_\ell \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}] \wedge \mathrm{r1Rf0}^i[\mathit{\Gamma}] \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}] \wedge \mathrm{r1Rf1}^\ell[\mathit{\Gamma}]) \qquad \wr\text{by invariant } S_{ind}(\mathit{\Gamma}, \Gamma)\wr \\
\Rightarrow & \mathsf{at}\{8\} \wedge \mathsf{at}\{28\} \wedge \big(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29} \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge (\mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma})\big) \vee \\
 & \big(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29}, i_9 \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge (\mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{9:},\ i_9,\ 1\rangle\rangle \in \mathit{\Gamma})\big) \vee \\
 & \big(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29}, i_{10} \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge (\mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{10:},\ i_{10},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{0:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma})\big) \vee \\
 & \big(\exists i, k_i, \ell, n_\ell, \ell_{30}, \ell_{29}, i_{10}, i_9 \in \mathbb{N} \;.\; \mathit{\Gamma} \in \Gamma \wedge (\mathfrak{rf}\langle L0^i_{k_i},\ \langle \mathtt{30:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F0^i,\ \langle \mathtt{29:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle L1^\ell_{n_\ell},\ \langle \mathtt{10:},\ i_{10},\ 1\rangle\rangle \in \mathit{\Gamma}) \wedge (\mathfrak{rf}\langle F1^\ell,\ \langle \mathtt{9:},\ i_9,\ 1\rangle\rangle \in \mathit{\Gamma})\big) \\
 & \quad \wr\text{def. } \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}], \mathrm{r1Rf0}^i[\mathit{\Gamma}], \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}], \text{ and } \mathrm{r1Rf1}^\ell[\mathit{\Gamma}], \mathfrak{rf}\langle x_\theta,\ \langle \ell\mathtt{:},\ \theta',\ v\rangle\rangle \text{ implies that } x_\theta = v, A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C), \exists \text{ distributes over } \vee, \text{ and } (\exists x \;.\; A(x)) \wedge B = \exists x \;.\; (A(x) \wedge B) \text{ when } x \text{ is not free in } B\wr \\
\Rightarrow & \mathsf{at}\{8\} \wedge \mathsf{at}\{28\} \wedge (\neg S_{com_1}(\mathit{\Gamma}, \Gamma) \vee \neg S_{com_2}(\mathit{\Gamma}, \Gamma) \vee \neg S_{com_3}(\mathit{\Gamma}, \Gamma) \vee \neg S_{com_4}(\mathit{\Gamma}, \Gamma)) \\
\Rightarrow & \neg S_{com}(\mathit{\Gamma}, \Gamma)
\end{array}
$$

by defining $S_{com}(\mathit{\Gamma}, \Gamma) \triangleq (\mathsf{at}\{8\} \wedge \mathsf{at}\{28\}) \Rightarrow (S_{com_1}(\mathit{\Gamma}, \Gamma) \wedge S_{com_2}(\mathit{\Gamma}, \Gamma) \wedge S_{com_3}(\mathit{\Gamma}, \Gamma) \wedge S_{com_4}(\mathit{\Gamma}, \Gamma))$ and the $S_{com_i}(\mathit{\Gamma}, \Gamma)$ as in Fig. 18.

Figure 19: Calculational design of S_{com} for PostgreSQL



counter-example to S_{com_1} counter-example to non-starvation

Figure 20: Counter-examples to PostgreSQL with WCM

Supported in part by NSF Grant CCF-1617717.

`{0: latch0 = 0; flag0 = 0; latch1 = 1; flag1 = 1; }`

1: $\{\mathit{\Gamma} \in \Gamma\}$
  `do` $\{i\}$
2: $\{\mathit{\Gamma} \in \Gamma\}$
   `do` $\{j_i\}$
3: $\{\mathit{\Gamma} \in \Gamma\}$
    `r[] Rl0 latch0` $\{\rightsquigarrow L0^i_{j_i}\}$
4: $\{\mathit{\Gamma} \in \Gamma \wedge \mathtt{Rl0} = L0^i_{j_i} \wedge (\mathrm{r0Rl0}^i_{j_i}[\mathit{\Gamma}] \vee \mathrm{r1Rl0}^i_{j_i}[\mathit{\Gamma}])\}$
   `while (Rl0=0)` $\{k_i\}$
5: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}]\}$
   `w[] latch0 0`
6: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}]\}$
   `r[] Rf0 flag0` $\{\rightsquigarrow F0^i\}$
7: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}] \wedge \mathtt{Rf0} = F0^i \wedge (\mathrm{r0Rf0}^i[\mathit{\Gamma}] \vee \mathrm{r1Rf0}^i[\mathit{\Gamma}])\}$
   `if (Rf0`$\neq$`0) then`
8: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}] \wedge \mathrm{r1Rf0}^i[\mathit{\Gamma}]\}$
    `(* critical section *)`
    `w[] flag0 0`
9: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}] \wedge \mathrm{r1Rf0}^i[\mathit{\Gamma}]\}$
    `w[] flag1 1`
10: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}] \wedge \mathrm{r1Rf0}^i[\mathit{\Gamma}]\}$
    `w[] latch1 1`
11: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl0}^i_{k_i}[\mathit{\Gamma}] \wedge \mathrm{r1Rf0}^i[\mathit{\Gamma}]\}$
   `fi`
12: $\{\mathit{\Gamma} \in \Gamma\}$
  `while true`
13: $\{\mathsf{false}\}$

||

21: $\{\mathit{\Gamma} \in \Gamma\}$
  `do` $\{\ell\}$
22: $\{\mathit{\Gamma} \in \Gamma\}$
   `do` $\{m_\ell\}$
23: $\{\mathit{\Gamma} \in \Gamma\}$
    `r[] Rl1 latch1` $\{\rightsquigarrow L1^\ell_{m_\ell}\}$
24: $\{\mathit{\Gamma} \in \Gamma \wedge \mathtt{Rl1} = L1^\ell_{m_\ell} \wedge (\mathrm{r0Rl1}^\ell_{m_\ell}[\mathit{\Gamma}] \vee \mathrm{r1Rl1}^\ell_{m_\ell}[\mathit{\Gamma}])\}$
   `while (Rl1=0)` $\{n_\ell\}$
25: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}]\}$
   `w[] latch1 0`
26: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}]\}$
   `r[] Rf1 flag1` $\{\rightsquigarrow F1^\ell\}$
27: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}] \wedge \mathtt{Rf1} = F1^\ell \wedge (\mathrm{r0Rf1}^\ell[\mathit{\Gamma}] \vee \mathrm{r1Rf1}^\ell[\mathit{\Gamma}])\}$
   `if (Rf1`$\neq$`0) then`
28: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}] \wedge \mathrm{r1Rf1}^\ell[\mathit{\Gamma}]\}$
    `(* critical section *)`
    `w[] flag1 0`
29: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}] \wedge \mathrm{r1Rf1}^\ell[\mathit{\Gamma}]\}$
    `w[] flag0 1`
30: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}] \wedge \mathrm{r1Rf1}^\ell[\mathit{\Gamma}]\}$
    `w[] latch0 1`
31: $\{\mathit{\Gamma} \in \Gamma \wedge \mathrm{r1Rl1}^\ell_{n_\ell}[\mathit{\Gamma}] \wedge \mathrm{r1Rf1}^\ell[\mathit{\Gamma}]\}$
   `fi`
32: $\{\mathit{\Gamma} \in \Gamma\}$
  `while true`
33: $\{\mathsf{false}\}$

Invariants:

$\mathrm{r0Rl0}^i_{j_i}[\mathit{\Gamma}] \triangleq (\mathfrak{rf}\langle L0^i_{j_i},\ \langle 0{:},\ \_,\ 0\rangle\rangle \in \mathit{\Gamma} \wedge L0^i_{j_i} = 0) \vee (\exists i_5 \in \mathbb{N}\ .\ \mathfrak{rf}\langle L0^i_{j_i},\ \langle 5{:},\ i_5,\ 0\rangle\rangle \in \mathit{\Gamma} \wedge L0^i_{j_i} = 0)$

$\mathrm{r1Rl0}^i_{j_i}[\mathit{\Gamma}] \triangleq (\exists \ell_{30} \in \mathbb{N}\ .\ \mathfrak{rf}\langle L0^i_{j_i},\ \langle 30{:},\ \ell_{30},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge L0^i_{j_i} = 1)$

$\mathrm{r0Rf0}^i[\mathit{\Gamma}] \triangleq (\mathfrak{rf}\langle F0^i,\ \langle 0{:},\ \_,\ 0\rangle\rangle \in \mathit{\Gamma} \wedge F0^i = 0) \vee (\exists i_8 \in \mathbb{N}\ .\ \mathfrak{rf}\langle F0^i,\ \langle 8{:},\ i_8,\ 0\rangle\rangle \in \mathit{\Gamma} \wedge F0^i = 0)$

$\mathrm{r1Rf0}^i[\mathit{\Gamma}] \triangleq (\exists \ell_{29} \in \mathbb{N}\ .\ \mathfrak{rf}\langle F0^i,\ \langle 29{:},\ \ell_{29},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge F0^i = 1)$

$\mathrm{r0Rl1}^\ell_{m_\ell}[\mathit{\Gamma}] \triangleq (\exists \ell_{25} \in \mathbb{N}\ .\ \mathfrak{rf}\langle L1^\ell_{m_\ell},\ \langle 25{:},\ \ell_{25},\ 0\rangle\rangle \in \mathit{\Gamma} \wedge L1^\ell_{m_\ell} = 0)$

$\mathrm{r1Rl1}^\ell_{m_\ell}[\mathit{\Gamma}] \triangleq (\mathfrak{rf}\langle L1^\ell_{m_\ell},\ \langle 0{:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma} \wedge L1^\ell_{m_\ell} = 1) \vee (\exists i_{10} \in \mathbb{N}\ .\ \mathfrak{rf}\langle L1^\ell_{m_\ell},\ \langle 10{:},\ i_{10},\ 1\rangle\rangle \in \mathit{\Gamma} \wedge L1^\ell_{m_\ell} = 1)$

$\mathrm{r0Rf1}^\ell[\mathit{\Gamma}] \triangleq (\exists m_{28} \in \mathbb{N}\ .\ \mathfrak{rf}\langle F1^\ell,\ \langle 28{:},\ m_{28},\ 0\rangle\rangle \in \mathit{\Gamma} \wedge F1^\ell = 0)$

$\mathrm{r1Rf1}^\ell[\mathit{\Gamma}] \triangleq (\mathfrak{rf}\langle F1^\ell,\ \langle 0{:},\ \_,\ 1\rangle\rangle \in \mathit{\Gamma} \wedge F1^\ell = 1) \vee (\exists i_9 \in \mathbb{N}\ .\ \mathfrak{rf}\langle F1^\ell,\ \langle 9{:},\ i_9,\ 1\rangle\rangle \in \mathit{\Gamma} \wedge F1^\ell = 1)$

Communications:

$\mathrm{RL0}^i_{j_i} \triangleq \{\mathfrak{rf}\langle L0^i_{j_i},\ \langle 0{:},\ \_,\ 0\rangle\rangle, \mathfrak{rf}\langle L0^i_{j_i},\ \langle 5{:},\ i_5,\ 0\rangle\rangle, \mathfrak{rf}\langle L0^i_{j_i},\ \langle 30{:},\ \ell_{30},\ 1\rangle\rangle \mid i_5 \in \mathbb{N} \wedge \ell_{30} \in \mathbb{N}\}$

$\mathrm{RF0}^i \triangleq \{\mathfrak{rf}\langle F0^i,\ \langle 0{:},\ \_,\ 0\rangle\rangle, \mathfrak{rf}\langle F0^i,\ \langle 8{:},\ i_8,\ 0\rangle\rangle, \mathfrak{rf}\langle F0^i,\ \langle 29{:},\ \ell_{29},\ 1\rangle\rangle \mid i_8 \in \mathbb{N} \wedge \ell_{29} \in \mathbb{N}\}$

$\mathrm{RL1}^\ell_{m_\ell} \triangleq \{\mathfrak{rf}\langle L1^\ell_{m_\ell},\ \langle 0{:},\ \_,\ 1\rangle\rangle, \mathfrak{rf}\langle L1^\ell_{m_\ell},\ \langle 25{:},\ \ell_{25},\ 0\rangle\rangle, \mathfrak{rf}\langle L1^\ell_{m_\ell},\ \langle 10{:},\ i_{10},\ 1\rangle\rangle \mid \ell_{25} \in \mathbb{N} \wedge i_{10} \in \mathbb{N}\}$

$\mathrm{RF1}^\ell \triangleq \{\mathfrak{rf}\langle F1^\ell,\ \langle 0{:},\ \_,\ 1\rangle\rangle, \mathfrak{rf}\langle F1^\ell,\ \langle 28{:},\ \ell_{28},\ 0\rangle\rangle, \mathfrak{rf}\langle F1^\ell,\ \langle 9{:},\ i_9,\ 1\rangle\rangle \mid \ell_{28} \in \mathbb{N} \wedge i_9 \in \mathbb{N}\}$

Anarchic communications:

$\overline{\Gamma} = \{\{\mathrm{rl0}^i_{j_i}, \mathrm{rf0}^i, \mathrm{rl1}^\ell_{m_\ell}, \mathrm{rf1}^\ell \mid i \in \mathbb{N} \wedge j_i \in [0, k_i] \wedge \ell \in \mathbb{N} \wedge j \in [0, n_\ell]\} \mid \forall i \in \mathbb{N}\ .\ \forall j_i \in [1, k_i]\ .\ \mathrm{rl0}^i_{j_i} \in \mathrm{RL0}^i_{j_i} \wedge \mathrm{rf0}^i \in \mathrm{RF0}^i \wedge \forall \ell \in \mathbb{N}\ .\ \forall m_\ell \in [1, m_\ell]\ .\ \mathrm{rl1}^\ell_{m_\ell} \in \mathrm{RL1}^\ell_{m_\ell} \wedge \mathrm{rf1}^\ell \in \mathrm{RF1}^\ell\}$

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