Hoare Logic for Multiprocessing

Marina Lenisa and Daniel Pellarini

UNIUD, Italy

marina.lenisa@uniud.it, daniel.pellarini@gmail.com

In the traditional approach of *Hoare logic*, the operational semantics of *concurrent programs* is explained via the *interleaving transition rule*. This reflects the execution on a single processor, where atomic actions of parallel components are interleaved and sequentially executed, and it does not directly account for multiprocessing systems.

Here we depart from the traditional approach, and we introduce a new *parallel operator* whose operational semantics directly reflects the execution on a *multiprocessing system*, where *disjoint* atomic actions of program components are executed in parallel.

Then, we develop a technique for verifying concurrent programs in this setting, inspired by the traditional Owicki-Gries method, see *e.g.* [ABO09]. In the multiprocessing setting, the above technique substantially simplifies, by only requiring *local interference freedom*, in place of *global interference freedom* between the proof outlines of parallel components. We plan to implement a tool for the verification of the local interference freedom; this could then be used in combination with a tool for verifying sequential components, such as *Why3* [BFMMP12], for verifying concurrent programs.

Coalgebraically (or game-theoretically), the new parallel operator can be interpreted as a kind of *conjunctive sum*, where each action consists of concurrent actions in more components. More in general, it would be interesting to explore what kind of parallelism corresponds to notions of sums alternative to the interleaving sum, which arise in the theory of coalgebras and games, [Con01, HL11, HLR11].

The Language.

We focus on the language for parallel programs with synchronization $\mathcal{L}_{par}$, whose syntax is defined as follows:

Definition 1 (Syntax).
$(\mathcal{L}_{par} \ni) S ::= skip \mid x := t \mid S_1;\ S_2 \mid await\ B\ then\ S\ end \mid if\ B\ then\ S_1\ else\ S_2\ fi \mid$
$while\ B\ do\ S_1\ od \mid [S_1 \mid\mid\mid \ldots \mid\mid\mid S_n]$

where

- $x, y, \ldots$ *are variables;*
- t *is an expression built over a standard language for integer and boolean expressions;*
- B *is a boolean expression;*
- *programs not containing the* $|||$ *operator are called* (sequential) components*;*
- *the program* S *in the* conditional atomic section $await\ B\ then\ S\ end$ *contains neither the* $|||$ *operator nor* $while$ *subprograms;*
- *the components* $S_1, \ldots, S_n$ *in the parallel composition* $[S_1 \mid\mid\mid \ldots \mid\mid\mid S_n]$ *do not contain the* $|||$ *operator.*

We assume an abstract level of granularity of our language, so as skip, assignment, and the evaluation of an if/while-guard are considered as *atomic actions*, as well as *conditional atomic sections*. The latter is executed in *mutual exclusion*, *i.e.* the guard B of a command $await\, B\, then\, S\, end$ is evaluated in the current state and, if it is true, then S is executed atomically with the evaluation of B, and in mutual exclusion.

Formally, the transition system of the language $\mathcal{L}_{par}$ is defined by a set of rules for deriving judgements of the shape $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle$, where $\sigma \in \Sigma$ is a *state*, *i.e.* a function from variables to values. The transition system that we consider includes the usual transition rules for the components (see [ABO09], Chapter 9), but it differs from the standard one because the rule for *interleaving* is replaced by the rule for *parallel execution of atomic actions*: at each step, in a parallel program $[S_1 \ ||| \ldots ||| \ S_n]$, a maximal set of *disjoint atomic actions* in the components is executed. Informally, two components, S_1 and S_2, execute *disjoint atomic actions* if the variables *modified* in the next atomic command/guard of each component are *not* used in the next atomic command/guard of the other. That is no written variable can be shared in the actions executed by the two components; conditional atomic regions cannot be executed in parallel with any other component.

In the following, for states σ, τ and X set of variables, we denote by $\sigma = \tau \ mod\ X$ the fact that the states σ and τ coincide on all variables but those in X.

Definition 2 (Parallel Transition Rule).
(i) Let $\tau_1 = \sigma \ mod\ X$, $\tau_2 = \sigma \ mod\ Y$, for $X \cap Y = \emptyset$. We define the state $\tau_1 \uplus \tau_2$ by

$$(\tau_1 \uplus \tau_2)(x) = \begin{cases} \sigma(x) & \text{if } x \notin X \cup Y \\ \tau_1(x) & \text{if } x \in X \\ \tau_2(x) & \text{if } x \in Y \end{cases}$$

(ii) Parallel transition rule:

$$\frac{\{< S_i,\ \sigma > \longrightarrow < S_i',\ \tau_i >\}_{i \in I}}{< [S_1 \ ||| \ldots \ ||| \ S_n],\ \sigma > \longrightarrow < [T_1 \ ||| \ldots \ ||| \ T_n],\ \biguplus_{i=1}^{n} \tau_i >}$$

where $\{S_i\}_{i \in I}$ is a maximal set of components executing disjoint atomic actions, and $T_i = \begin{cases} S_i & \text{if } i \notin I \\ S_i' & \text{if } i \in I\ . \end{cases}$

With the above parallel transition rule we make two implicit assumptions. First, we assume that the number of processors available is not bounded, or at least not less than the maximum number of components which can execute disjointly. Possibly, one can impose a limitation on the number of components which can execute in parallel, according to the number of processors, but this will not change the theory, and hence, for simplicity, we work without this assumption. The second implicit assumption is that all atomic actions, being executed in parallel on different processors, have the same "cost" in terms of execution time.

Notice that, the final states generated by computations arising with the interleaving rule are in general more than those induced by computations arising with the parallel transition rule. Namely, with the latter, *maximal* sets of disjoint atomic actions are forced to be executed at each step, and hence not all interleaving executions are compatible. This will be exploited in order to simplify the verification of parallel programs.

Verification of Concurrent Programs.

The standard technique due to Owicki-Gries for the verification of concurrent programs is based on the construction of *standard proof outlines* for the components, *i.e.* proof outlines where each atomic command or atomic region is preceded by exactly one assertion, and on the control of *interference-freedom* between these proof outlines. Intuitively, this latter step substantially simplifies in our multiprocessing setting, because of the parallelism constraints. In the following, we propose a technique for verifying concurrent programs in our setting. After having built standard proof outlines for components in the usual way, we proceed as follows:

1. We build the graph of *abstract computations* of the parallel program; by an *abstract computation* we mean a computation where we forget about the states, and we only account for the sequence of programs that we reach by executing the original program, and for the atomic actions/conditional section executed at each step (see Definition 3 below). Each node in the graph of abstract computations represents a point in the parallel program reached during its execution, and it is labeled by the sequence of corresponding assertions annotating the proof outlines of the components. The arcs in the graph will be labeled by the sequence of atomic commands/guards or by the conditional atomic action executed at that step. Notice that the graph has a finite number of nodes.
2. Once the graph of abstract computations has been built, the proof of *interference freedom* between the proof outlines of the components reduces to a *local* check of non interference between the assertions labeling a node and the sequence of atomic actions or the atomic section labeling the outgoing arcs.

The graph of abstract computations.

Definition 3 (Abstract Transition System and Computation).

(i) The abstract transition system *consists of rules for deriving judgements $S \xrightarrow{l} S'$, where l is a label representing (a sequence of) atomic commands/guards/sections or the empty action ε (representing a computation that doesn't perform any action in that specific computation step),* i.e.*:*

$$l \ ::= \ skip \mid x := t \mid B \mid \varepsilon \mid await\ B\ then\ S\ end \mid \langle l_1, \ldots, l_n \rangle \ .$$

The abstract transition rules are the following:

$$\frac{}{skip \xrightarrow{skip} E} \qquad \frac{}{x := t \xrightarrow{x:=t} E} \qquad \frac{}{S \xrightarrow{\varepsilon} S}$$

$$\frac{}{await\ B\ then\ S\ end \xrightarrow{await\ B\ then\ S\ end} E} \qquad \frac{S_1 \xrightarrow{l_1} S_1'}{S_1; S_2 \xrightarrow{l_1} S_1'; S_2}$$

$$\frac{}{if\ B\ then\ S_1\ else\ S_2\ fi \xrightarrow{B} S_1} \qquad \frac{}{if\ B\ then\ S_1\ else\ S_2\ fi \xrightarrow{\neg B} S_2}$$

$$\frac{}{while\ B\ do\ S\ od \xrightarrow{\neg B} E} \qquad \frac{}{while\ B\ do\ S\ od \xrightarrow{B} S; while\ B\ do\ S\ od}$$

$$\frac{\{S_i \xrightarrow{l_i} S_i'\}_{i\in I}}{[S_1' \,|||\ldots|||\, S_n] \xrightarrow{<l_1',\ldots,l_n'>} [S_1' \,|||\ldots|||\, S_n']}$$

where $\{S_i\}_{i\in I}$ *is a maximal set of components executing disjoint atomic actions and* $l_i' = \begin{cases} l_i & \text{if } i \in I \\ \varepsilon & \text{if } i \notin I\ . \end{cases}$

(ii) An abstract computation *is a (finite or infinite) sequence* $S \xrightarrow{l_1} S_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} S_n \ldots$

Notice that in all abstract computations, even the infinite ones, only finitely many different programs can appear.

Now we sketch how to define the (finite rooted) graph representing the abstract computations generating from a program $S = [S_1|||\ldots|||S_n]$. Each node n represents a point in the computation of S, and it is labeled by the n-tuple of assertions $\langle p^1_{j_1}, \ldots, p^n_{j_n}\rangle$ appearing in the proof outlines at that point. The construction of the graph starts from the root n, which is labeled with the initial assertions, and proceeds by analyzing, for each created node n', the abstract transitions arising from the corresponding program $[S_1' \,|||\ldots|||\, S_n']$: for each transition $[S_1' \,|||\ldots|||\, S_n'] \xrightarrow{<l_1,\ldots,l_n>} [S_1'' \,|||\ldots|||\, S_n'']$, a new node n'' is built, if it does not already exists, corresponding to $[S_1'' \,|||\ldots|||\, S_n'']$, and an arc is drawn from n' to n'', labeled by $\langle l_1, \ldots, l_n\rangle$. Some optimizations can be performed during the graph construction, by avoiding to represent transitions corresponding to the evaluation of a guard which is not compatible with the current assertions.

Local interference freedom. Once the graph of abstract computations has been built, the non-interference checks can be performed at a local level. That is, for any node n, for any outgoing arc and any atomic action appearing in its label and in the proof outline of a component, it is sufficient to check that this atomic action does not interfere with the assertions of the node n, which appear in the proof outlines of other components.

Coalgebraically, this new parallel operator can be interpreted as a kind of *conjunctive sum*, where each action consists of concurrent actions in more components. This is currently being studied as a natural continuation of this work.

References

[ABO09] K. Apt, F. de Boer, E. Olderog. *Verification of Sequential and Concurrent Programs*, Springer, 2009.

[BFMMP12] F. Bobot, J-C. Filliâtre, C. Marché. G. Melquiond, A. Paskevich, *The Why3 Platform*, Version 0.72, May 2012, available at http://why3.lri.fr/#documentation.

[Con01] J.H. Conway. On Numbers and Games, A K Peters Ltd, 2001.

[HL11] F. Honsell, M. Lenisa. Conway Games, algebraically and coalgebraically, *Logical Methods in Computer Science* **7(3)**, 2011.

[HLR11] F. Honsell, M. Lenisa, R. Redamalla. Equivalences and Congruences on Infinite Conway Games, *Theoretical Informatics and Applications* **46(2)**, 231–259, 2012.