Acta Informatica 24, 1–31 (1987)

Acta Informatica

Sometime = Always + Recursion ≡ Always on the Equivalence of the Intermittent and Invariant Assertions Methods for Proving Inevitability Properties of Programs

Patrick Cousot[1] and Radhia Cousot[2]

[1] Ecole Polytechnique, Centre de Mathématiques Appliquées, F-91128 Palaiseau Cedex, France
[2] Université de Paris-Sud, Centre d'Orsay, LRI, Bat. 490, F-91405 Orsay Cedex, France

Summary. We propose and compare two induction principles called "always" and "sometime" for proving inevitability properties of programs. They are respective formalizations and generalizations of Floyd invariant assertions and Burstall intermittent assertions methods for proving total correctness of sequential programs whose methodological advantages or disadvantages have been discussed in a number of previous papers. Both principles are formalized in the abstract setting of arbitrary nondeterministic transition systems and illustrated by appropriate examples. The "sometime" method is interpreted as a recursive application of the "always" method. Hence "always" can be considered as a special case of "sometime". These proof methods are strongly equivalent in the sense that a proof by one induction principle can be rewritten into a proof by the other one. The first two theorems of the paper show that an invariant for the "always" method can be translated into an invariant for the "sometime" method even if every recursive application of the later is required to be of finite length. The third and main theorem of the paper shows how to translate an invariant for the "sometime" method into an invariant for the "always" method. It is emphasized that this translation technique follows the idea of transforming recursive programs into iterative ones. Of course, a general translation technique does not imply that the original "sometime" invariant and the resulting "always" invariant are equally understandable. This is illustrated by an example.

1. Introduction

We compare two induction principles (that we call "always" and "sometime") for proving inevitability properties of transition systems, the nondeterminism of which can be unbounded. These induction principles are formalizations of program proof methods independently of a particular programming language, of a particular inevitability property and of a particular style of presentation

of proof methods. Thanks to these abstract and concise formalizations we can make more rigorous comparisons of program proof methods which for some time have always been compared using methodological arguments based on examples [8, 12].

We introduce the induction principle called "always" in [4]. It is a generalization of those program proof methods in the class of Floyd [7] invariant assertions method for proving total correctness of sequential programs. We introduce the induction principle called "sometime" in [5]. It is an abstract generalization of Burstall [2] intermittent assertions method for proving total correctness of sequential programs.

The "always" induction principle involves an induction along execution traces whereas the "sometime" induction principle involves a combination of an induction along (parts of) execution traces (connected to Burstall's "hand simulation") and recursion induction (related to Burstall's "induction on the data"). Hence the "always" induction principle corresponds to the particular case of the "sometime" induction principle when recursion is not used.

The "always" and "sometime" induction principles are sound and semantically complete (see the proofs respectively in [4, 5]) hence weakly equivalent in the sense that when a proof exists by one method, a proof must exist by the other method. Here we prove a stronger equivalence result, in the sense that whenever a proof exists by one method it can be rewritten into a proof by the other method. Intuitively this is because recursion can be eliminated from proofs in favour of induction in just the same way as it can be eliminated from programs in favour of iteration.

We illustrate our techniques by examples.

2. Inevitability Properties of Programs Represented as Transition Systems

To introduce abstract formalizations of program proof methods, we view programs as (nondeterministic) transition systems $\langle S, t \rangle$ [9] where S is a nonempty set of states and the transition relation t between a state and its possible successors is represented as a function $t \in (S \times S \to \{tt, ff\})$ from pairs of states into truth values (tt is true and ff is false).

We say that the nondeterminism of t is *bounded* when states can only have a finite number of successor states that is to say when $\forall s \in S. |\{s' \in S: t(s, s')\}| \in \omega$ (where $|E|$ is the cardinal of a class E and ω is the set of natural numbers) and *unbounded* otherwise. There is no need for the nondeterminism of t to be bounded. Hence we can represent parallel programs and even fair ones as transition systems by decomposition of these programs into their atomic actions. This decomposition usually destroys syntactic structure of programs and is therefore a debatable way of proceeding. However we neither suggest that program proofs should make a direct use of our induction principles nor that programs should be represented as transition systems before initiating the proofs. On the contrary we attach a great importance to the presentation of proofs. In [3] we have proposed a systematic method for constructing proof methods from induction principles, so that, in particular, the structural information about

the original program need not be lost in the proof method (although it might have been in case of direct use of the induction principle).

Example 2.1 (*The transition system corresponding to the tree traversing program*). The following sequential program to traverse a binary tree and count its tips is taken (literally) from Burstall [2]. It will be used as example throughout the paper.

The value of variable Tr of type tree is either nil or (lf(Tr).rg(Tr)) where lf(Tr) and rg(Tr) are trees, the value of Co of type nat is a natural number and the value of variable St of type stack is either ( ) or (hd(St).tl(St)) where hd(St) is a tree and tl(St) is a stack.

```
Start: St:=( ); Co:=0;
Loop: if Tr≠nil
        then begin Push Tr onto St;
                   Tr:=lf(Tr); goto Loop
             end
        else begin Co:=Co+1;
                   if St=( ) then goto Finish;
                   Pop Tr from St;
                   Tr:=rg(Tr); goto Loop
             end;
Finish:
```

We will use capital letters Tr, ... for program variables and small letters possibly with quotes tr, tr′, tr″, ... for their values at different time instants.

The transition system $\langle S, t\rangle$ corresponding to that program is defined by:

$$S = \{\text{Start, Loop, Finish}\} \times \text{tree} \times \text{nat} \times \text{stack}$$

$$\begin{aligned} t(\langle l', \text{tr}', \text{co}', \text{st}'\rangle, \langle l'', \text{tr}'', \text{co}'', \text{st}''\rangle) = \\ [(l' = \text{Start} \wedge l'' = \text{Loop} \wedge \text{tr}'' = \text{tr}' \wedge \text{co}'' = 0 \wedge \text{st}'' = ( )) \\ \vee (l' = \text{Loop} \wedge \text{tr}' \neq \text{nil} \wedge l'' = \text{Loop} \wedge \text{tr}'' = \text{lf}(\text{tr}') \wedge \text{co}'' = \text{co}' \wedge \text{st}'' = (\text{tr}'.\text{st}')) \\ \vee (l' = \text{Loop} \wedge \text{tr}' = \text{nil} \wedge \text{st}' \neq ( ) \wedge l'' = \text{Loop} \wedge \text{tr}'' = \text{rg}(\text{hd}(\text{st}')) \wedge \\ \text{co}'' = \text{co}' + 1 \wedge \text{st}'' = \text{tl}(\text{st}')) \\ \vee (l' = \text{Loop} \wedge \text{tr}' = \text{nil} \wedge \text{st}' = ( ) \wedge l'' = \text{Finish} \wedge \text{tr}'' = \text{tr}' \wedge \text{co}'' = \text{co}' + 1 \\ \wedge \text{st}'' = \text{st}')]. \quad \square \end{aligned}$$

Executions of a program $\langle S, t\rangle$ shall be modelled by its set $\Sigma\langle S, t\rangle$ of complete execution traces $p_0, p_1, p_2, \ldots$ where the p_i are states, there is a transition from p_i to p_{i+1} and the sequence is either infinite or terminates in a state p_n which has no possible successor. More formally,

— 0 is the empty set or zero,
— If $n \in \omega$ and $n \neq 0$ then n will denote $\{0, \ldots n-1\}$,
(so that $m \in n$ is equivalent to $m < n$)

— If E is a set then $E \sim x = \{y \in E : y \neq x\}$,
— $D \rightarrow\!\!\!\!\!\rightarrow R$ is the class of partial functions from D into R,
— $D \rightarrow R$ is the class of total functions from D into R,
— $\Sigma^0\langle S, t\rangle = 0$
Empty traces are not considered,
— $\Sigma^n\langle S, t\rangle = \{p \in (n \rightarrow S) : \forall i \in (n-1).\, t(p_i, p_{i+1}) \wedge \forall s \in S.\, \neg t(p_{n-1}, s)\}$
Finite traces of length $n > 0$ (p_i is short for $p(i)$),
— $\Sigma^\omega\langle S, t\rangle = \{p \in (\omega \rightarrow S) : \forall i \in \omega.\, t(p_i, p_{i+1})\}$
Infinite traces,
— $\Sigma\langle S, t\rangle = \bigcup_{n \in \omega} \Sigma^n\langle S, t\rangle \cup \Sigma^\omega\langle S, t\rangle$
Complete traces.

A relation $\psi \in (S \times S \rightarrow \{tt, ff\})$ is inevitable for a program $\langle S, t\rangle$ if any program execution eventually leads to a state that is related to the initial state by ψ.

More formally, $\psi \in (S \times S \rightarrow \{tt, ff\})$ is *inevitable* for $\langle S, t\rangle$ if and only if

$$\forall p \in \Sigma\langle S, t\rangle.\, \exists i \in Dom(p).\, \psi(p_0, p_i) \tag{1}$$

(where $Dom(p) = \{x : \exists y.(y = p(x))\}$ is the domain of function p).

Example 2.2 (*Total correctness as an inevitability property*). Total correctness of program 2.1 can be specified by the inevitability of ψ such that:

$$\psi((l, \mathrm{tr}, \mathrm{co}, \mathrm{st}), (l', \mathrm{tr}', \mathrm{co}', \mathrm{st}')) = [(l = \mathrm{Start}) \Rightarrow (l' = \mathrm{Finish} \wedge \mathrm{co}' = \mathrm{tips}(\mathrm{tr}))]$$

where

$$\mathrm{tips}(\mathrm{tr}) = \textit{if } \mathrm{tr} = \mathrm{nil} \textit{ then } 1 \textit{ else } \mathrm{tips}(\mathrm{lf}(\mathrm{tr})) + \mathrm{tips}(\mathrm{rg}(\mathrm{tr})). \quad \square$$

Extending Dijkstra's definitions of strong and weak termination [6] (see also Back [1], Gries [8]), we say that inevitability is *strong* when the number i of program steps necessary for reaching the "final" state p_i is bounded by an integer k depending only on the "initial" state p_0.

More formally, $\psi \in (S \times S \rightarrow \{tt, ff\})$ is *strongly inevitable* for $\langle S, t\rangle$ if and only if

$$\forall s \in S.\, \exists k \in \omega.\, \forall p \in \Sigma\langle S, t\rangle.\, [(p_0 = s) \Rightarrow (\exists i \leqq k.\, \psi(p_0, p_i))]$$

$\psi \in (S \times S \rightarrow \{tt, ff\})$ is *weakly inevitable* for $\langle S, t\rangle$ if and only if ψ is inevitable for $\langle S, t\rangle$ but not strongly.

3. The "Always" Induction Principle Generalizing Floyd's Invariant Assertions Proof Method

Floyd's total correctness proof method for sequential programs [7] was generalized to inevitability proofs for nondeterministic transition systems $\langle S, t\rangle$ and formalized by a number of strongly equivalent induction principles in Cousot

and Cousot [4], the most abstract one being:

$$[\exists \Gamma \in Ord, J \in (\Gamma \times S \times S \to \{tt, ff\}). \tag{2}$$

(F.1) $\quad (\forall s \in S . \exists \gamma \in \Gamma . J(\gamma, s, s))$

(F.2) $\quad \wedge (\forall s, s' \in S, \gamma' \in \Gamma .$

$\quad J(\gamma', s, s') \Rightarrow$

(F.2.a) $\quad [(\exists s'' \in S . t(s', s'') \wedge \forall s'' \in S . [t(s', s'') \Rightarrow \exists \gamma'' < \gamma' . J(\gamma'', s, s'')])$

(F.2.b) $\quad \vee \psi(s, s')])]$

where $(Ord, <)$ is the class of ordinals.

This induction principle is better understood by first considering the case when the nondeterminism of t is bounded. Then one can choose $\Gamma = \omega$ (instead of $\Gamma \in Ord$) in induction principle (2) and prove strong inevitability. More precisely, the induction hypothesis $J(\gamma', s, s')$ is true when "current" state s' is a descendant of "initial" state s and from s' on, execution will lead in at most γ' steps to some "final" state s'' satisfying $\psi(s, s'')$. In particular by F.1, for any initial state s there is an integer γ such that execution for this initial state s is guaranteed to lead to some final state (s'' satisfying $\psi(s, s'')$) in less than γ steps. Moreover by F.2, a descendant s' of an initial state s which is not a final state must have a successor state s'' and all its successor states must be closer to the goal ψ.

We now show that induction principle (2) is a model of Floyd's "always" proof method using the following:

Example 3.1 (*Proof of the tree traversing program using the "always" induction principle*). 1. Floyd's method requires three separate proofs to establish total correctness, one to show partial correctness, the other to show termination and the last to show absence of run-time errors.

(i) The partial correctness proof of program 2.1 first consists in discovering intermediate assertions attached to the control points of the program. These assertions should be local invariants that is to say express relationships that hold between the initial values tr, co, st of variables Tr, Co, St and the values tr', co', st' of these variables Tr, Co, St whenever control passes through the corresponding points:

$$I_{\text{Start}}(\text{tr}, \text{co}, \text{st}, \text{tr}', \text{co}', \text{st}') = [\text{tr}' = \text{tr} \wedge \text{co}' = \text{co} \wedge \text{st}' = \text{st}]$$

$$I_{\text{Loop}}(\text{tr}, \text{co}, \text{st}, \text{tr}', \text{co}', \text{st}') = [(\text{tips}(\text{tr}') + \text{co}' + \mathit{Sum}(\text{tips} \circ \text{rg}, \text{st}')) = \text{tips}(\text{tr})]$$

$$I_{\text{Finish}}(\text{tr}, \text{co}, \text{st}, \text{tr}', \text{co}', \text{st}') = [\text{co}' = \text{tips}(\text{tr})]$$

where

$$f \circ g(x) = f(g(x)) \quad \text{and if } f \in (\text{tree} \to \omega) \text{ and st is a stack then}$$

$$\mathit{Sum}(f, \text{st}) = \mathit{if}\ \text{st} = (\,)\ \mathit{then}\ 0\ \mathit{else}\ f(\text{hd}(\text{st})) + \mathit{Sum}(f, \text{tl}(\text{st})).$$

Observe that these local invariants on values of program variables are equivalent to the following global invariant on program states:

$$I(\langle l, \text{tr}, \text{co}, \text{st} \rangle, \langle l' \text{tr}', \text{co}', \text{st}' \rangle) = [(l = \text{Start}) \Rightarrow I_{l'}(\text{tr}, \text{co}, \text{st}, \text{tr}', \text{co}', \text{st}')]$$