Exploiting Historical Information about Computations *

UF CISE Technical Report 96-034

Beverly A. Sanders † K. Mani Chandy‡

Abstract

This paper explores the question: How can we exploit the information that a state predicate p holds at some point in a computation in reasoning about the remainder of the computation? This question is relevant because partial global snapshots of distributed systems can determine that a predicate holds at a point in the computation, even though the partial snapshot does not determine the complete global state. This question is also relevant in extending the substitution axiom of UNITY.

Keywords: concurrency, program correctness, program specification UNITY.

1 Introduction

A state predicate p of a program F is *stable* if and only if: for all state transitions (s, t) of F, if p holds in s then p holds in t as well. Therefore, for any stable predicate p of a program F and any state s of F, if p holds in s then p holds in all states reachable from s.

If a stable predicate p holds at some point x in a computation, then p continues to hold for the remainder of the computation. Informally speaking, when we reason about the remainder of a computation from point x onwards, we can restrict universal quantification to mean quantification over states for which p holds because all other states are unreachable; so, in reasoning about the remainder of the computation we can substitute *true* for p (and vice versa). This idea is made precise in this paper.

*Supported in part by U.S. Air Force Office of Scientific Research and the University of Florida

†Department of Computer and Information Science and Engineering, P.O. Box 116120, University of Florida, Gainesville, FL 32611-6120, sanders@cise.ufl.edu.

‡Department of Computer Science, California Institute of Technology, m/c 256-80, Pasadena, CA 91125, mani@cs.caltech.edu.

In UNITY, the fact that some states are unreachable is handled with the substitution axiom: An invariant can be replaced by *true*, and vice versa, in any program property. All states reachable from initial states satisfy invariants. Just as the initial condition is helpful in reasoning about all computations, a predicate that holds at a point in a computation is helpful in reasoning about the remainder of the computation.

We propose a class of program properties, called *since* properties after Manna and Pnueli [10]. The interpretation of $@p : X$, read as "since p X, " is that if a stable state predicate p holds at any point in any computation, then property X holds from that point onwards in that computation. (A property is a predicate on computations; *always* p, *eventually* p, and p *next* q, are examples [10, 9, 2].)

After reviewing some preliminary results, we give a general definition and theory of *since* properties. An important requirement is that the theory be applicable in composed systems where one deduces the properties of a system from the properties of its components. To this end, we develop the rules in the context of the theory of composed systems presented in [3, 4].

2 Preliminaries

A program is a tuple (V, T), where V is a set of typed variables defining a state space Σ_V and T is a set of commands where T includes *skip*, and the weakest-precondition predicate transformers for each command is universally-conjunctive and or-continuous. Including *skip* in T has the effect of allowing "stuttering" steps that leave the state unchanged in computations. The restrictions on the predicate transformer ensure that the corresponding commands always terminate, and have demonic and bounded non-determinism.

A computation of a program is an infinite sequence of pairs (c_i, s_i) for $i \geq 0$ where $c_i \in T$ and $s_i \in \Sigma_V$ and c_i represents a command that can take state s_i to state s_{i+1}. Initial conditions are discussed later. Progress properties usually depend on some sort of fairness assumption: For the purposes of this paper, we assume that all commands must be executed infinitely often in every computation.

Since at each step in a computation, any command can be executed next, the conjunction of the weakest preconditions of the commands gives the weakest precondition of a computation step. We have [3]:

$$[awp.F.q \quad \equiv \quad (\forall t : t \in T : t.q)]. \tag{1}$$

Thus if at some point in a program, the state satisfies $awp.F.q$, then after the next step, the state satisfies q. Since $wp.skip$ is the identity, $[awp.F.q \Rightarrow q]$ for all q. Since $wp.s$ is required to be universally conjunctive for all s, so is awp.

We define a function *stable* from state predicates to program properties next. For a state predicate p of a program F, $stable.p$ is a program property, and hence

stable.p.F is a boolean. It is defined as:

$$stable.q.F = \quad [awp.F.q \equiv q]. \tag{2}$$

Thus if q is stable predicate of a program F, and q holds at some point in a computation, then q continues to hold in the remainder of that computation.

It has been shown [13] that conjunction and disjunction of stable predicates are stable, and that the set of stable predicates of a program forms a lattice where *false* is the strongest stable predicate and *true* is the weakest.

3 Since properties

3.1 Properties defined by $[q \Rightarrow f.r]$

Several program properties of interest can be defined in terms of a predicate transformer f as follows:

$$q \xrightarrow{f} r = [q \Rightarrow f.r] \tag{3}$$

The above notation is used for convenience, the following table lists several predicate transformers along with their more common notation and operational interpretation.

next	*awp*	p *next* q	if p holds, then q holds after the next step	[12, 3]
'leads-to'	*wlt*	$p \rightsquigarrow q$	if p holds, then eventually, at that point or later q holds	[11, 8, 7]
'to-stable'	*wto*	$p \rightarrow q$	if p holds, then eventually $\neg q$ becomes unreachable	[3, 6]
'to-always'	*wta*	$p \hookrightarrow q$	if p holds, then $\neg q$ will hold at most a finite number of steps	[1, 6]

All the predicate transformers f in this table satisfy the following formula that we call a "healthiness condition:" For the program F under consideration: f is monotonic and satisfies

$$stable.p.F \quad \Rightarrow \quad [p \wedge f.q \Rightarrow f.(p \wedge q)] \tag{4}$$

We define a *since* version of the property determined by f as

$$@p: \; q \xrightarrow{f} r = stable.p \wedge [p \wedge q \Rightarrow f.r] \tag{5}$$

Operationally, $@p:\ q \xrightarrow{f} r$ means that if p holds at some point in a computation, then the property $q \xrightarrow{f} r$ holds for the remainder of that computation.

Several rules follow immediately from the definition, For instance,

$$stable.p \Rightarrow ((@p':\ q \xrightarrow{f} r) \Rightarrow (@p \wedge p':\ q \xrightarrow{f} r))$$

In addition to these rules *since* properties admit a substitution rule analogous to the UNITY substitution axiom. The essence of the idea is that if p holds at some point in a computation of a program then p can be replaced by *true* in properties of the remainder of the computation. Next, we present a precise definition of this notion.

Substitution rule Let p,q, and r be punctual[1] and f be a monotonic predicate transformer satisfying (4).

$$\begin{aligned} @p:\ q.p \xrightarrow{f} r.p &= @p:\ q.true \xrightarrow{f} r.p \\ &= @p:\ q.p \xrightarrow{f} r.true \\ &= @p:\ q.true \xrightarrow{f} r.true \end{aligned} \tag{6}$$

(7)

Proof From the definition and predicate calculus, it suffices that the following two conditions hold;

$$[p \Rightarrow (q.p \equiv q.true)] \tag{8}$$

$$[p \Rightarrow (f.(r.p) \equiv f.(r.true))] \tag{9}$$

The first condition (8) is simply a restatement of the assumption of punctuality. The second (9) requires a proof since f is typically not a punctual function of its arguments.

$$\begin{array}{ll} & [p \ \Rightarrow \ (f.(q.p) \equiv f.(q.true))] \\ = & \qquad \{ \text{ predicate calculus } \} \\ & [p \wedge f.(q.p) \ \Rightarrow f.(q.true)] \ \wedge \ [p \wedge f.(q.true) \ \Rightarrow f.(q.p)] \\ \Leftarrow & \qquad \{ \ stable \ \ p, \ f \text{ satisfies (4 } \} \\ & [f.(q.p \wedge p) \ \Rightarrow \ f.(q.true)] \quad \wedge \quad [f.(q.true \wedge p) \ \Rightarrow \ f.(q.p)] \\ \Leftarrow & \qquad \{ \ f \text{ monotonic } \} \\ & [q.p \wedge p \ \Rightarrow \ q.true] \quad \wedge \quad [q.true \wedge p \ \Rightarrow \ q.p] \end{array}$$

[1] Recall that a function q is punctual if for all x and y, $[(x \equiv y) \Rightarrow (q.x \equiv q.y)]$. The punctuality theorem [5] states that expressions built from variables and punctual operators are punctual functions of the variables. It is also the case that the logical connectives ($\wedge, \vee, \Rightarrow, \Leftarrow, \neg, \equiv$) are punctual in their arguments.

$=$

$\quad [p \Rightarrow (q.true \equiv q.p)]$

$\Leftarrow$ $\quad \{\ q$ punctual $\}$

$\quad true$

The condition (4) obviously holds for awp and also has been shown to hold for *wlt*, *wto*, and *wta*. Indeed, we postulate it as a "healthiness condition" for predicate transformers used to define temporal properties.

3.2 Transient predicates

Transient predicates are important as they form the basis of practical proof rules for leads-to, to-stable, and to-always properties [11, 4, 6]. Transient does not, however, have the shape given in the previous section and is thus treated separately here. We define a function *transient* from state predicates to program properties as follows. For a program $F = (V, T)$,

$$transient.q.F \quad \equiv \quad (\exists t : t \in T : [q \Rightarrow wp.t.\neg q]) \tag{10}$$

with the operation interpretation that *transient* q means that $\neg q$ will hold infinitely often in any computation.

The *since* property for transient is

$$@p :\ transient.q.F \quad \equiv \quad stable.p \wedge (\exists t : t \in T : [p \wedge q \Rightarrow wp.t.\neg q]) \tag{11}$$

with the operational interpretation that if stable predicate p holds in a computation, then $\neg q$ will hold infinitely often in that computation. Since *wp.s* satisfies (4) and $\neg$ preserves punctuality, a proof similar to the one given for (9) establishes, for punctual q, that

$$@p :\ transient\ q.p = @p :\ transient\ q.true$$

thus extending the substitution rule for transient properties.

3.3 Properties of since properties

We list a few fairly obvious facts below. X, Y, and Z are properties.

- For programs with unspecified initial conditions

  $$X = @true :\ X$$

- For programs with specified initial conditions and strongest invariant SI [13]:

  $$X = @SI :\ X$$

- $\quad$ *stable* $p \wedge @q :\ X \quad \Rightarrow \quad @q \wedge p :\ X$
- If $(\forall F : X.F \wedge Y.F \Rightarrow Z.F)$ then

$$@p :\ X \wedge @p :\ Y \quad \Rightarrow \quad @p :\ Z$$

4 Parallel Composition

4.1 Definitions

In this section, we define parallel composition in our model and briefly review an approach to component specifications presented in [3, 4]. Then, we give rules for since properties of components.

For $F = (V_F, T_F)$ and $G = (V_G, T_G)$, the parallel composition of F and G is given by

$$F \parallel G \quad = \quad (V_F \cup V_G,\ T_F \cup T_G,) \tag{12}$$

and is defined if all variables appearing in both V_F and V_G have the same type. From the definition, $\parallel$ is commutative and associative.

We define a function *component* so that *component*.$G.F$ holds for a program F if G is a component of F. From the definition of parallel composition (12),

$$\textit{component}.G.F \equiv (V_G \subseteq V_F \ \ \wedge \ \ T_G \subseteq T_F).$$

Viewing properties as predicates on programs, we have a property transformer wg [3] defined as follows

$$wg.F.X \quad \equiv \quad (\forall H : \textit{component}.F.H \Rightarrow X.H)$$

with the interpretation that $wg.F.X$ is the weakest property so that if a program containing F as a component satisfies that property, then the program satisfies property X. In analogy with the way the predicate transformer f admitted a property, we have

$$\begin{aligned} (X\ \textit{guarantees} Y).F \quad &= \quad (X \stackrel{wg}{\rightarrow} Y).F \\ &= \quad (X \Rightarrow wg.F.X) \end{aligned} \tag{13}$$

Thus $(X \ \ \textit{guarantees} \ \ Y).F$ means that if property X holds for a program containing F as a component, then property Y also holds for the program.[2] Given $(X \ \ \textit{guarantees} \ \ Y).F$ as a part of the specification of F, after proving $X.F \| G$, we can conclude $Y.F \| G$. This is useful when X is more easily proved than Y.

[2] Note that this does *not* say that if property X holds for an environment E that Y holds for $F \| E$.

4.2 Compositional specification with all-component and exist component properties

Two classes of program properties have useful special characteristics:

All-component properties Property X is called an *all-component* property if for some set of programs $\mathcal{G}$,

$$(\forall G : G \in \mathcal{G} : X.G) \quad = \quad (\|G : G \in \mathcal{G} : G).X$$

An all-component property can be proved for a system by showing that it holds for each component.

Exist-component properties A property X is called an *exist-component* property if for any set of programs $\mathcal{G}$,

$$(\exists G : G \in \mathcal{G} : G.X) \quad = \quad (\|G : G \in \mathcal{G} : G).X$$

For an exists-component property X,

$$X.F \quad \Rightarrow (true = (wg.F.X).F \| G)$$

Of the properties mentioned earlier, in our model *next* ($\overset{awp}{\rightarrow}$) and stable are all-component properties, *transient* is an exist component property, and all proofs of 'leads-to', ($\overset{wlt}{\rightarrow}$), 'to-stable' ($\overset{wto}{\rightarrow}$), and 'to-alway' ($\overset{wta}{\rightarrow}$) can be proved as combinations of all-component and exist-component properties.

The following theorems follow easily from the definitions:

All-component since properties If X is an all-component property, then $@p : X$ is an all-component property.

Exist-component since properties If $(@p : X).F$ and X is an exist-component, then

$$(stable \ \ p) \quad \Rightarrow (wg.F.(@p : X))$$

or alternatively,

$$((stable \ \ p) \quad guarantees \quad (@p : X)).F$$

5 Partial Global Snapshots

A global snapshot records the global state of a computation. A partial global snapshot records a predicate that holds at some point in a computation. For instance, a partial global snapshot may determine that there is at least one token in a distributed system, or that processes F and G are waiting. The

question we explore next is: How can we exploit knowledge that a predicate p holds at a point in a computation?

We define a predicate transformer *sst*, for strongest stable, as follows [13]: $sst.p$ is the strongest stable predicate weaker than p. Therefore, $sst.p$ holds exactly in those states reachable from states for which p holds. So, if p holds at a point in a computation, then $sst.p$ holds from that point onwards.

If a program F has the property $@p' : X$, and $[p \Rightarrow p']$, then if p holds at a point in a computation of F, then X is a property of the remainder of the computation from that point onwards. Similarly, if $@sst.p : X$ holds, then if p holds at a point in the computation of F, then X is a property of the remainder of the computation. Thus, partial global snapshots can be used to reason about the computation from the snapshot point onwards.

6 Conclusion

Rather than taking advantage of knowledge of only of the initial state of a computation, we have proposed a class of properties that take into account knowledge of the state at any point in a computation. Not only are these properties more expressive, but they offer a more practical alternative to the 'substitution axiom' for composed systems.

References

[1] K. Mani Chandy. Properties of parallel programs. *Formal Aspects of Computing*, 1993.

[2] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[3] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24:129–148, 1995.

[4] K. Mani Chandy and Beverly A. Sanders. Reasoning about program composition. *submitted for publication*, 1996.

[5] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[6] Rutger M. Dijkstra and Beverly A. Sanders. A predicate transformer for the progress property 'to-always'. *submitted for publication*, 1996.

[7] C.S. Jutla, E. Knapp, and J.R. Rao. A predicate transformer approach to semantics of parallel programs. In *Proceeding of the 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[8] Edgar Knapp. A predicate tranformer for progress. *Information Processing Letters*, 33, 1989/90.

[9] Leslie Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[10] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

[11] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[12] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[13] Beverly Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2), 1991.