SRI International

CSL Technical Report ● September 2001

Formal Verification of McMillan's Compositional Assume-Guarantee Rule

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NASA Langley Research Center under contract NAS1-20334 and Cooperative Agreement NCC-1-377 with Honeywell Tucson, and by the DARPA MoBIES program under contract F33615-00-C-1700 with US Air Force Research Laboratory.

Abstract

To illustrate some of the power and convenience of its specification language and theorem prover, we use the PVS formal verification system to verify the soundness of a proof rule for assume-guarantee reasoning due to Ken McMillan.

Contents

Chapter 1

Introduction

The key idea in assume-guarantee reasoning, first introduced by Chandy and Misra [MC81] and Jones [Jon83], is that we show that component X_1 guarantees certain properties P_1 on the assumption that component X_2 delivers certain properties P_2, and *vice versa* for X_2, and then claim that the composition of X_1 and X_2 (i.e., both running and interacting together) guarantees P_1 and P_2 unconditionally.

We can express this idea symbolically in terms of the following putative proof rule.

$$\frac{\begin{array}{c}\langle P_2 \rangle X_1 \langle P_1 \rangle \\ \langle P_1 \rangle X_2 \langle P_2 \rangle\end{array}}{\langle true \rangle \;\; X_1 || X_2 \;\; \langle P_1 \wedge P_2 \rangle}$$

Here, $X_1||X_2$ denotes the composition of X_1 and X_2 and formulas like $\langle p \rangle X \langle q \rangle$ assert that if X is part of a system that satisfies p (i.e., p is true of all behaviors of the composite system), then the system must also satisfy q (i.e., X assumes p and guarantees q).

Rules such as this are called "compositional" because we reason about X_1 and X_2 separately (in the hypotheses above the line) and deduce properties about $X_1||X_2$ (in the conclusion below the line) without having to reason about the composed system directly. The problem with such proof rules is that they are circular (X_1 depends on X_2 and *vice versa*) and potentially unsound.

In fact, the unsoundness is more than potential, it is real: for example, let P_1 be "eventually $x = 1$," let P_2 be "eventually $y = 1$," let X_1 be "wait until $y = 1$, then set x to 1," and let X_2 be "wait until $x = 1$, then set y to 1," where both x and y are initially 0. Then the hypotheses to the rule are true, but the conclusion is not: X_1 and X_2 can forever wait for the other to make the first move.

There are several modified assume-guarantee proof rules that are sound. Different rules may be compared according to the kinds of system models and specification they support, the extent to which they lend themselves to mechanized analysis, and the extent to which they are preserved under refinement (i.e., the circumstances under which X_1 can be replaced by an implementation that may do more than X_1). Early work considered many different system models for the components—for example, (terminating) programs that communicate by shared variables or by synchronous or asynchronous message passing—while the properties considered could be those true on termination (e.g., input/output relations), or characterizations of the conditions under which termination would be achieved. Later work considers the components as reactive systems (i.e., programs that maintain an ongoing interaction with their environment) that interact through shared variables, and whose properties are formalized in terms of their behaviors (i.e., roughly speaking, the sequences of values assumed by their state variables).

One way to obtain a sound compositional rule is to break the "circular" dependency in the previous one by introducing an intermediate property I such that

$$\frac{\begin{array}{c}\langle P_1\rangle X_1 \langle I\rangle \\ \langle I\rangle X_2 \langle P_2\rangle\end{array}}{\langle P_1\rangle \;\; X_1 || X_2 \;\; \langle P_2\rangle.}$$

The problem with this approach is that "circular" dependency is a real phenomenon and cannot simply be legislated away. In the Time Triggered Architecture (TTA) [KG94, TTT01], for example, clock synchronization depends on group membership and group membership depends on synchronization—so we do need a proof rule that truly accommodates this circularity. However, closer examination of the circular dependency in TTA reveals that it is not circular if the temporal evolution of the system is taken into consideration: clock synchronization in round t depends on group membership in round $t-1$, which in turn depends on clock synchronization in round $t-2$ and so on.

This suggests that we could modify our previous circular rule to read as follows, where P_j^t indicates that P_j holds up to time t.

$$\frac{\begin{array}{c}\langle P_2^t\rangle X_1 \langle P_1^{t+1}\rangle \\ \langle P_1^t\rangle X_2 \langle P_2^{t+1}\rangle\end{array}}{\langle \mathit{true}\rangle \;\; X_1 || X_2 \;\; \langle P_1 \wedge P_2\rangle}$$

Although this seems intuitively plausible, we really want the t and $t+1$ on the same side of each antecedent formula, so that we are able to reason from one time point to the next. A formulation that has this character has been introduced by McMillan [McM99]; here H is a "helper" property (which can be simply *true*), $\Box$ is the "always" modality of Linear Temporal Logic (LTL), and $p \rhd q$ (" p constrains q") means that if p is always true up to time t, then q holds at time $t+1$ (i.e., p fails before q).

$$\frac{\begin{array}{c}\langle H\rangle X_1 \langle P_2 \rhd P_1\rangle \\ \langle H\rangle X_2 \langle P_1 \rhd P_2\rangle\end{array}}{\langle H\rangle \;\; X_1 || X_2 \;\; \langle \Box(P_1 \wedge P_2)\rangle} \tag{1.1}$$

Notice that $p \rhd q$ can be written as the LTL formula $\neg(p \,\mathsf{U}\, \neg q)$, where U is the LTL "until" operator.[1] This means that the antecedent formulas can be established by LTL model checking if the transition relations for X_1 and X_2 are finite.

The proof rule 1.1 has the characteristics we require, but what exactly does it mean, and is it sound? These question can be resolved only by giving a semantics to the symbols and formulas used in the rule. McMillan's presentation of the rule only sketches the argument for its soundness; a more formal treatment is given by Namjoshi and Trefler [NT00], but it is not easy reading and does not convey the basic intuition.

Accordingly, we present in the next Chapter a formalization and verification of McMillan's rule using PVS. The development is surprisingly short and simple and should be clear to anyone with knowledge of PVS.

[1]The subexpression $p \,\mathsf{U}\, \neg q$ holds if q eventually becomes false, and p was true at every preceding point; this is the exact opposite of what we want, hence the outer negation.

Chapter 2

Formalization and Verification in PVS

We begin with a PVS datatype that defines the basic language of LTL (to be interpreted over a state type `state`).

```
pathformula[state : TYPE]: DATATYPE
BEGIN
  Holds(state_formula: pred[state]): Holds?
  U(arg1: pathformula, arg2: pathformula): U?
  X(arg: pathformula): X?
  ~(arg: pathformula): NOT?
  \/(arg1: pathformula, arg2: pathformula): OR?
END pathformula
```

Here, `U` and `X` represent the *until* and *next* modalities of LTL, respectively, and `~` and `\/` represent negation and disjunction, respectively. `Holds` represents application of a state (as opposed to a path) formula.

The semantics of the language defined by `pathformula` are given by the function `|=` defined in the theory `paths`. LTL formulas are interpreted over sequences of states (thus, an LTL formula specifies a set of such sequences). The definition `s |= P`[1] (`s` satisfies `P`) recursively decomposes the pathformula `P` by cases and determines whether it is satisfied by the sequence `s` of states.

[1]PVS infix operators such as `|=` must appear in prefix form when they are defined.

```
paths[state: TYPE]: THEORY
BEGIN
  IMPORTING pathformula[state]

  s: VAR sequence[state]
  P, Q : VAR pathformula

  |=(s,P): RECURSIVE bool =
     CASES P OF
      Holds(S) : S(s(0)),
      U(Q, R): EXISTS (j:nat): (suffix(s,j) |= R) AND
                        (FORALL (i: below(j)): suffix(s,i) |= Q),
      X(Q): rest(s) |= Q,
      ~(Q) : NOT (s |= Q),
      \/(Q, R): (s |= Q) OR (s |= R)
    ENDCASES
  MEASURE P by <<
```

The various cases are straightforward. A state formula `S` `Holds` on a sequence `s` if it is true of the first state in the sequence (i.e., `s(0)`). `U(Q, R)` is satisfied if some suffix of `s` satisfies `R` and `Q` was satisfied at all earlier points. `X(Q)` is satisfied by `s` if `Q` is satisfied by the rest of `s`. The functions `suffix` and `rest` (which is equivalent to `suffix(1)`) are defined in the PVS prelude. Negation and disjunction are defined in the obvious ways.

Given semantics for the basic operators of LTL, we can define the other operators in terms of these.

```
  CONVERSION+ K_conversion

  <>(Q) : pathformula = U(Holds(TRUE), Q) ;
  [](Q) : pathformula = ~<>~Q  ;
  &(P, Q) : pathformula = ~(~P \/ ~Q) ;
  =>(P, Q) : pathformula = ~P \/ Q ;
  <=>(P, Q) : pathformula = (P => Q) & (Q => P) ;
  |>(P, Q): pathformula = ~(U(P,~Q))

END paths
```

Here `<>` and `[]` are the *eventually* and *always* modalities, respectively. A formula `Q` is eventually satisfied by `s` if it is satisfied by some suffix of `s`. The `CONVERSION+` command turns on PVS's use of K Conversion (named after the K combinator of combinatory logic), which is needed in the application of `U` in the `<>` construction to "lift" the constant `TRUE` to a predicate on states. The *constrains* modality introduced by McMillan is specified as `|>`.

We are less interested in interpreting LTL formulas over arbitrary sequences of states than over those sequences of states that are generated by some system or program. We specify programs as transition relations on states; a state sequence `s` is then a *path* (or *trace*) of a program (i.e., it represents a possible sequence of the states as the program executes) if each pair of adjacent states in the sequence is consistent with the transition relation. These notions are specified in the theory `assume_guarantee`, which is parameterized by a `state` type and a transition relation `N` over that type.

```
assume_guarantee[state: TYPE, N: pred[[state, state]]]: THEORY
BEGIN
  IMPORTING paths[state]

  i, j: VAR nat
  s: VAR sequence[state]

  path: TYPE = {s | FORALL i: N(s(i), s(i + 1))}
  p: VAR path
  JUDGEMENT suffix(p, i) HAS_TYPE path
```

A key property, expressed as a PVS judgement (i.e., a lemma that can be applied by the typechecker) is that every suffix to a path of `N` is also a path of `N`. The proof obligation that justifies this judgement is proved automatically.

Next, we specify what it means for a pathformula `P` to be *valid* for `N` (this notion is not used in this development, but it is important in others).[2] We then state a useful lemma `and_lem`. It is proved by `(GRIND)`.

```
  H, P, Q: VAR pathformula

  valid(P): bool = FORALL p: p |= P

  and_lem: LEMMA (p |= (P & Q)) = ((p |= P) AND (p |= Q))
```

Next, we define the function `ag(P, Q)` that gives a precise meaning to the informal notation `<P> N <Q>` used earlier (again, the `N` is implicit as it is a theory parameter).

```
  ag(P, Q): bool = FORALL p: (p |= P) IMPLIES (p |= Q)
```

Two key lemmas are then stated and proved.

```
  agr_box_lem: LEMMA ag(H, []Q) =
    FORALL p, i: (p |= H) IMPLIES (suffix(p,i) |= Q)

  constrains_lem: LEMMA ag(H, P |> Q) =
    FORALL p, i: (p |= H)
      AND (FORALL (j: below(i)): suffix(p, j) |= P)
         IMPLIES (suffix(p, i) |= Q)
END assume_guarantee
```

[2]Note that `N` is implicit as it is a parameter to the theory; this is necessary for the `JUDGEMENT`, which would otherwise need to contain `N` as a free variable (which is not allowed in the current version of PVS).

The first lemma allows the *always* (`[]`) modality to be removed from the conclusion of an assume-guarantee assertion, while the second lemma allows elimination of the *constrains* (`|>`) modality. Both of these are proved by `(GRIND :IF-MATCH ALL)`.

Finally, we can specify composition and McMillan's rule for compositional assume-guarantee reasoning.

```
composition[state: TYPE]: THEORY
BEGIN
  N, N1, N2: VAR PRED[[state, state]]

  //(N1, N2)(s, t: state): bool =  N1(s, t) AND N2(s, t)

  IMPORTING assume-guarantee

  i, j: VAR nat
  H, P, Q: VAR pathformula[state]

  kens_thm: THEOREM
       ag[state, N1](H, P |> Q) AND ag[state, N2](H, Q |> P)
          IMPLIES
            ag[state, N1//N2](H, [](P & Q))

END composition
```

Here, `//` is an infix operator that represents composition of programs, defined as the conjunction of their transition relations. Then, `kens_thm` is a direct transliteration into PVS of the proof rule 1.1 on page 4. The PVS proof of this theorem is surprisingly short: it basically uses the lemmas to expose and index into the paths, and then performs a strong induction on that index.

```
 (SKOSIMP)
 (APPLY (REPEAT
    (THEN (REWRITE "agr_box_lem") (REWRITE "constrains_lem"))))
 (INDUCT "i" :NAME "NAT_induction")
 (SKOSIMP* :PREDS? T)
 (REWRITE "and_lem[state,N1!1 // N2!1]")
 (GROUND)
 (("1" (APPLY (THEN (INST -6 "p!1" "j!1") (LAZY-GRIND))))
  ("2" (APPLY (THEN (INST -5 "p!1" "j!1") (LAZY-GRIND)))))
```

Our first attempt to formalize this approach to assume-guarantee reasoning was long, and the proofs were also long—and difficult. Other groups have apparently invested months of work in a similar endeavor without success. That the final treatment in PVS is so straightforward is testament to the expressiveness of the PVS language (e.g., its ability to define LTL in a few dozen lines) and the power and integration of its prover (e.g., the predicate subtype `path` and its associated `JUDGEMENT`, which automatically discharges numerous side conditions during the proof).

Although we have proved McMillan's assume-guarantee method to be sound, it is known to be incomplete (i.e., there are correct systems that cannot be verified using the rule 1.1). Namjoshi and Trefler [NT00] present an extended rule that is both sound and complete, and it would be interesting to extend our PVS verification to this rule.

Another extension would expand the formal treatment from the two-process to the n-process case (this is a technical challenge in formal verification, rather than an activity that would yield additional insight).

Finally, it will be useful to investigate practical application of the approach presented here. One possible application is to the mutual interdependence of membership and synchronization in TTA: each of these is verified on the basis of assumptions about the other.

Acknowledgment

The PVS formalization of LTL was performed by Carsten Schürmann. The PVS formalization and proof of 1.1 was a collaborative effort with Jonathan Ford, Sam Owre, Harald Rueß, and N. Shankar.

Bibliography

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983. 3

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994. 4

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981. 3

[McM99] K. L. McMillan. Circular compositional reasoning about liveness. In Laurence Pierre and Thomas Kropf, editors, *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 342–345, Bad Herrenalb, Germany, September 1999. Springer-Verlag. 4

[NT00] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 139–153, Chicago, IL, July 2000. Springer-Verlag. 5, 11

[TTT01] Time-Triggered Technology TTTech Computertechnik AG, Vienna, Austria. *Specification of the TTP/C Protocol (version 0.6p0504)*, May 2001. 4