Compositional Specification and Reasoning

Yih-Kuen Tsay

Dept. of Information Management

National Taiwan University

Outline

- The Very Beginning: Pre and Post-Conditions
- Concurrency: Taking Interference into Account
- Compositional Methods
- The Mutual Induction Mechanism
- Assume-Guarantee Specification in LTL
- Interface Automata etc.
- Concluding Remarks

Pre and Post-Conditions

This seminal paper started it all:

C.A.R. Hoare. An axiomatic basis for computer programs. *CACM*, 12(8):576-580, 1969.

- Notation: $\{P\}\ S\ \{Q\}$ (originally, $P\ \{S\}\ Q$)
  Meaning: If the execution of S starts in a state satisfying P and terminates, then it results in a state satisfying Q.
- Proof rules (and axioms) were developed in accordance with the interpretation.

Note: R.W. Floyd did something similar for flowcharts earlier in 1967, which was also a precursor of “proof outline”.

Sequential Composition

- Rule of (Sequential) Composition:

$$\frac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

- Rule of Consequence:

$$\frac{P \rightarrow P',\ \{P'\}\ S\ \{Q'\},\ Q' \rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Adaptations of these rules would allow one to talk about replacing one sequential component by another.

Sequential vs. Concurrent Components

- Both generate computations, which are sequences of states possibly with labels on the steps: $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \cdots \xrightarrow{l_n} s_n \; (\xrightarrow{l_{n+1}} s_{n+1} \xrightarrow{l_{n+2}} \cdots)$.
- For a sequential component, only its start and final states matter to other components.
- Computations of a concurrent component are produced by *interleaving its steps with those of an 'arbitrary but compatible' environment*.
- Many interesting concurrent components, often referred to as *reactive* components, are not meant to terminate.

Taking Interference into Account

Probably the first and best-known attempt at generalizing Hoare Logic to concurrent programs is:

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

- Proof outlines (for terminating programs)
- Interference Freedom
- Auxiliary variables

Interference Freedom

A proof outline Δ_1 is not interfered by a second proof outline Δ_2 if the following holds.

- Let P_1 be an arbitrary assertion in Δ_1.
- Let $\{P_2\}\ S_2\ \{Q_2\}$ be an arbitrary triple in Δ_2.
- It is true that $\{P_1 \wedge P_2\}\ S_2\ \{P_1\}$.

Two proof outlines are interference-free if none of them interferes with the other.

Criteria of Compositionality

- Compositional specifications of a component should not refer to the internal structures of itself and/or other components.
- This is desirable, as we often want to speak of replacing a component by another that satisfies the same specification.
- So, Owicki and Greis' method does not qualify as a compositional method.

Remark: Owicki and Greis' method (or its adaptation) is probably the most usable when one has at hand all the code of a (small) concurrent system.

Lamport's 'Hoare Logic'

In this probably forgotten paper, Lamport proposed a new interpretation to pre and post-conditions:

L. Lamport. The 'Hoare Logic' of concurrent programs. *Acta Informatica*, 14:21-37, 1980.

- Notation: $\{P\}\ S\ \{Q\}$
  Meaning: If execution starts anywhere in S with P true, then executing S (1) will leave P true while control is in S and (2) if terminating, will make Q true.
- The usual Hoare triple would be expressed as $\{P\}\ \langle S \rangle\ \{Q\}$, where $\langle \cdot \rangle$ indicates atomic execution.

Lamport's 'Hoare Logic' (cont.)

- Rule of consequence (can't strengthen the pre-condition):

$$\frac{\{P\}\ S\ \{Q'\},\ Q' \to Q}{\{P\}\ S\ \{Q\}}$$

- Rules of Conjunction and Disjunction:

$$\frac{\{P\}\ S\ \{Q\},\ \{P'\}\ S\ \{Q'\}}{\{P \wedge P'\}\ S\ \{Q \wedge Q'\}} \qquad \frac{\{P\}\ S\ \{Q\},\ \{P'\}\ S\ \{Q'\}}{\{P \vee P'\}\ S\ \{Q \vee Q'\}}$$

Lamport's 'Hoare Logic' (cont.)

- Rule of Sequential Composition:

$$\frac{\{P\}\ S\ \{Q\},\ \{R\}\ T\ \{U\},\ Q \wedge at(T) \rightarrow R}{\{(in(S) \rightarrow P) \wedge (in(T) \rightarrow R)\}\ S;T\ \{U\}}$$

- Rule of Parallel Composition:

$$\frac{\{P\}\ S_i\ \{P\},\ 1 \leq i \leq n}{\{P\}\ \textbf{cobegin}\ \mathop{\|}_{i=1}^{n} S_i\ \textbf{coend}\ \{P\}}$$

UNITY Logic

UNITY was once quite popular. Its logic has been modified in a subsequent work.

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

- A program consists of (1) an initial condition and (2) a set of actions (or conditional multiple-assignments), which always includes skip.
- Main Notation: $p\,co\,q \triangleq \forall s :: \{p\}\ s\ \{q\}$ (over all action s of a given program).

Note: There are also operators for liveness properties.

UNITY Logic (cont.)

- Notation: $p\,co\,q \triangleq \forall s :: \{p\}\ s\ \{q\}$ (p constrains q)
- Meaning: Whenever p holds, q holds after the execution of any single action (if it terminates).
- Examples:
  - "$\forall m :: x = m \ \ co \ \ x \geq m$" says x never decreases.
  - "$\forall m, n :: x, y = m, n \ \ co \ \ x = m \vee y = n$" says x and y never change simultaneously.

UNITY Logic vs. 'Hoare Logic'

- "co" enjoys the complete rule of consequence.
- Rules of conjunction and disjunction also hold.
- Stronger rule of parallel composition:

$$\frac{p\; co\; q \text{ in } F,\;\; p\; co\; q \text{ in } G}{p\; co\; q \text{ in } F \,\|\, G}$$

- But, "co" is much less convenient for sequential composition.

Jones' Rely/Guarantee Pairs

C.B. Jones. Tentative steps towards a development method for interfering programs. *TOPLAS*, 5(4):596-619, 1983.

- Assumption about the environment is expressed by a pre-condition and a *rely*-condition
- Promised behavior of a component is expressed by a post-condition and a *guarantee*-condition.
- Both rely and guarantee-conditions are predicates of two states, to deal with reactive behavior.

We will illustrate rely and guarantee-conditions in the context of temporal logic.

Assume-Guarantee Specification

A component will behave properly only if its environment (the context where it is used) does.

To summarize the lessons learned, the specification of a component should include

1. assumed properties about its environment and
2. guaranteed properties of the module if the environment obeys the assumption.

We will focus on reactive behavior from now on.

Mutual Dependency

Let $A \rhd G$ denote a generic component specification with assumption A and guarantee G.

The following composition rule looks plausible, but is circular and unsound without an adequate semantics for $\rhd$.

$$\frac{\begin{array}{l} [\![M_1]\!] \models A_1 \rhd G_1 \\ [\![M_2]\!] \models A_2 \rhd G_2 \\ A \wedge G_1 \rightarrow A_2 \\ A \wedge G_2 \rightarrow A_1 \end{array}}{[\![M_1 \parallel M_2]\!] \models A \rhd (G_1 \wedge G_2)}$$

The circularity may be broken by introducing a mutual induction mechanism into $\rhd$.

The Mutual Induction Mechanism

The mechanism was probably first proposed in

> J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

- Notation: $r \mid h \mid s$
  - h is a CSP-like process with message communication.
  - r and s are assertions on the *traces* of h
- Meaning: (1) s holds initially and (2) if r holds up to the k-th point in a trace of h, then s holds up to the $(k+1)$-th point in that trace, for all k.

Note: "$r[h]s$" is used if r or s also refers to the internal communication channels of h.

Misra and Chandy's Proof System

- Rule of network composition:

$$\frac{r_i \mid h_i \mid s_i,\ 1 \leq i \leq n}{(\bigwedge_{i=1}^{n} r_i)[\, \|_{i=1}^{n} h_i](\bigwedge_{i=1}^{n} s_i)}$$

- Rule of inductive consequence:

$$\frac{(s \wedge r) \rightarrow r';\quad r' \mid h \mid s}{r \mid h \mid s} \qquad \frac{r \mid h \mid s';\quad s' \rightarrow s}{r \mid h \mid s}$$

Misra and Chandy's Proof System (cont.)

- Theorem of Hierarchy:

$$\frac{r_i \mid h_i \mid s_i,\ 1 \leq i \leq n;\ (\bigwedge_{i=1}^{n} s_i \wedge R_0) \rightarrow \bigwedge_{i=1}^{n} r_i;\ \bigwedge_{i=1}^{n} s_i \rightarrow S_0}{R_0 \mid \mathop{\|}_{i=1}^{n} h_i \mid S_0}$$

There are also rules for proving "$r \mid h \mid s$" from scratch.

Limit of the Mutual Induction Mechanism

- Induction on the length of computation works for safety properties (invariants).
- But, it does not for liveness, which needs explicit well-founded induction (by defining variant functions that decrease as computation progresses)

Modular Reasoning in Temporal Logic

A. Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and Models of Concurrent Systems*, 123-144. Springer, 1985.

- Steps by the component and those by its environment need to be distinguished.
- Induction structures are required.
- Computations of a component allow arbitrary environment steps
- Past temporal operators (as an alternative to history variables) are useful.
- Barringer and Kuiper had explored some of the above ideas earlier [LNCS 197, 1984].

Conditions for Easy Compositionality

- Exactly one single component is accountable for changes at the interface in each step.
- Input-enabled: a component is always ready to perform any input action (which is paired with some output action from the environment).
  - For shared-variable models, this is automatically true.
- With these conditions, $[\![C_1 \parallel C_2]\!]$ can be easily understood as $[\![C_1]\!] \cap [\![C_2]\!]$.

Modular Reasoning in TLA

The probably most-cited work of assume-guarantee specification in temporal logic is:

M. Abadi and L. Lamport. Conjoining specifications. *TOPLAS*, 17(3):507-534, 1995.

- Main notation: $E \stackrel{+}{\rightarrow\!\!\triangleright} M$
  Meaning: (1) M holds initially and (2) for $n \geq 0$, if E holds for the prefix of length n in a computation, then M holds for the prefix of length $n + 1$.
- TLA is extended in some sense.
- Liveness properties are treated.

Modular Reasoning in LTL

However, we will describe instead our own work, which is similar:

B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167:47-72, 1996.

- It makes good use of past temporal operators.
- Proof rules are purely syntactical in LTL.

Note: We will omit the treatment of hiding and liveness.

LTL

An LTL formula is interpreted over an infinite sequence of states $\sigma = s_0, s_1, s_2, \ldots, s_i, \ldots$ relative to a position.

- State formulae: $(\sigma, i) \models \varphi$ iff φ holds at s_i.
- $(\sigma, i) \models \bigcirc\varphi$ (“next φ”) iff $(\sigma, i+1) \models \varphi$.
- $(\sigma, i) \models \Box\varphi$ (“henceforth φ”) iff $\forall k \geq i : (\sigma, k) \models \varphi$.
- $(\sigma, i) \models \circleddash\varphi$ (“before φ”) iff $(i > 0) \rightarrow ((\sigma, i-1) \models \varphi)$.
- $(\sigma, i) \models \boxminus\varphi$ (“so-far φ”) iff $\forall k : 0 \leq k \leq i : (\sigma, k) \models \varphi$.

$\neg\varphi$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\ldots$, etc. are defined in the obvious way. We will not use $\Diamond$ or $\diamondminus$ in this talk.

LTL (cont.)

Syntactic sugars:

- u^- denotes the value of u in the previous state; by convention, u^- equals u at position 0.
- $first \triangleq \circleddash false$, which holds only at position 0.

A sequence σ is *satisfies* a temporal formula φ if $(\sigma, 0) \models \varphi$.

A formula φ is *valid*, denoted $\models \varphi$, if φ is satisfied by every sequence.

Program KEEP-AHEAD

local a, b : **integer where** $a = b = 0$

$$P_a :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\ a := b + 1\ \right] \end{array} \right] \parallel P_b :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\ b := a + 1\ \right] \end{array} \right]$$

$$(a = 0) \wedge (b = 0) \wedge \Box \left(\begin{array}{cl} & (a = b^- + 1) \wedge (b = b^-) \\ \vee & (b = a^- + 1) \wedge (a = a^-) \\ \vee & (a = a^-) \wedge (b = b^-) \end{array} \right)$$

Program KEEP-AHEAD(cont.)

local a, b : **integer where** $a = b = 0$

$$P_a :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\ a := b + 1\ \right] \end{array} \right] \parallel P_b :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\ b := a + 1\ \right] \end{array} \right]$$

$$\Box \left((\mathit{first} \rightarrow (a = 0) \land (b = 0)) \land \left(\begin{array}{ll} & (a = b^- + 1) \land (b = b^-) \\ \lor & (b = a^- + 1) \land (a = a^-) \\ \lor & (a = a^-) \land (b = b^-) \end{array} \right) \right)$$

Modularized Program KEEP-AHEAD

module M_a **in** b : **integer** **out** a : **integer** $= 0$ **loop forever do** $\left[\ a := b + 1\ \right]$	$\|\|$	**module** M_b **in** a : **integer** **out** b : **integer** $= 0$ **loop forever do** $\left[\ b := a + 1\ \right]$

Modularized Program KEEP-AHEAD (cont.)

$$\Phi_{M_a} \triangleq (a = 0) \wedge \Box \begin{pmatrix} & (a = b^- + 1) \wedge (b = b^-) \\ \vee & (a = a^-) \end{pmatrix}$$

$$\Phi_{M_b} \triangleq (b = 0) \wedge \Box \begin{pmatrix} & (b = a^- + 1) \wedge (a = a^-) \\ \vee & (b = b^-) \end{pmatrix}$$

Parallel Composition as Conjunction

- The parallel composition of modules M_a and M_b is equivalent to Program KEEP-AHEAD; formally,

$$\Phi_{M_a} \wedge \Phi_{M_b} \;\leftrightarrow\; \Phi_{\text{KEEP-AHEAD}}\,.$$

- Let Φ_M denote the system specification of a module M. We take $\Phi_M \rightarrow \varphi$ as the formal definition of the fact that M satisfies φ, also denoted as $M \models \varphi$.
- If M is a module of system S (i.e., $S \equiv M \wedge M'$, for some M'), then $M \models \varphi$ implies $S \models \varphi$.

Assume-Guarantee Formulae

- Assume that the assumption and the guarantee are safety formulae respectively of the forms $\Box H_A$ and $\Box H_G$, where H_A and H_G are past formulae (containing no future temporal operators).
- An A-G formula is defined as follows:

$$\Box H_A \triangleright \Box H_G \triangleq \Box(\circleddash \boxminus H_A \rightarrow \boxminus H_G)$$

or equivalently,

$$\Box H_A \triangleright \Box H_G \triangleq \Box(\circleddash \boxminus H_A \rightarrow H_G).$$

- Note 1: $\Box H_A \triangleright \Box H_G$ implies H_G holds initially (at position 0).
- Note 2: $(true \triangleright \Box H_G) \equiv \Box H_G$.

Refinement

- Refinement of Guarantee

$$\frac{\Box[\widetilde{\ominus}\boxminus H_A \wedge \boxminus H_{G'} \rightarrow \boxminus H_G]}{\Box(\widetilde{\ominus}\boxminus H_A \rightarrow \boxminus H_{G'}) \rightarrow \Box(\widetilde{\ominus}\boxminus H_A \rightarrow \boxminus H_G)}$$

- Refinement of Assumption

$$\frac{\Box[\boxminus H_A \wedge \boxminus H_A \rightarrow \boxminus H_{A'}]}{\Box(\widetilde{\ominus}\boxminus H_{A'} \rightarrow \boxminus H_G) \rightarrow \Box(\widetilde{\ominus}\boxminus H_A \rightarrow \boxminus H_G)}$$

Composing A-G Specifications

$$\models\ (\Box H_{G_1} \triangleright \Box H_{G_2}) \wedge (\Box H_{G_2} \triangleright \Box H_{G_1}) \ \rightarrow\ \Box H_{G_1} \wedge \Box H_{G_2}.$$

This shows that A-G formulae have a mutual induction mechanism built in and hence permit “circular reasoning” (mutual dependency).

Composing A-G Specifications (cont.)

Suppose that $\Box H_{A_i}$ and $\Box H_{G_i}$, for $1 \leq i \leq n$, $\Box H_A$, and $\Box H_G$ are safety formulae.

$$\begin{array}{ll} 1. & \models \Box\Big(\boxminus H_A \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_{A_j}\Big), \text{ for } 1 \leq j \leq n \\ 2. & \models \Box\Big(\circleddash \boxminus H_A \wedge \boxminus \bigwedge_{i=1}^{n} H_{G_i} \rightarrow H_G\Big) \\ \hline & \models \bigwedge_{i=1}^{n} (\Box H_{A_i} \rhd \Box H_{G_i}) \rightarrow (\Box H_A \rhd \Box H_G) \end{array}$$

A Compositional Verification Rule

Rule MOD-S:
Suppose that A_i, G_i, and G are canonical safety formulas.
Then,

$$\frac{\begin{array}{l} [\![M_i]\!] \models A_i \rhd G_i \text{ for } 1 \leq i \leq n \\ \bigwedge_{i=1}^{n} (A_i \rhd G_i) \to G \end{array}}{[\![\mathop{\|}_{i=1}^{n} M_i]\!] \models G}$$

A Compositional Proof

Let us try to verify $[\![\text{KEEP-AHEAD}]\!] \models \Box((a \geq a^-) \wedge (b \geq b^-))$ compositionally:

The composition rule suggests decomposing $\Box((a \geq a^-) \wedge (b \geq b^-))$ as the conjunction of

$$\Box(b \geq b^-) \ \triangleright\ \Box(a \geq a^-) \text{ and } \Box(a \geq a^-) \ \triangleright\ \Box(b \geq b^-).$$

Unfortunately, neither $[\![M_a]\!] \models \Box(b \geq b^-) \ \triangleright\ \Box(a \geq a^-)$ nor $[\![M_b]\!] \models \Box(a \geq a^-) \ \triangleright\ \Box(b \geq b^-)$.

A Compositional Proof (cont.)

A simple remedy is to first strengthen the proof obligation as $[\![\text{KEEP-AHEAD}]\!] \models \Box((\mathit{first} \rightarrow a \geq 0) \wedge (a \geq a^-) \wedge (\mathit{first} \rightarrow b \geq 0) \wedge (b \geq b^-))$.

The composition rule again suggests a similar decomposition:

$$\Box((\mathit{first} \rightarrow b \geq 0) \wedge b \geq b^-) \;\triangleright\; \Box((\mathit{first} \rightarrow a \geq 0) \wedge a \geq a^-)$$

and

$$\Box((\mathit{first} \rightarrow a \geq 0) \wedge a \geq a^-) \;\triangleright\; \Box((\mathit{first} \rightarrow b \geq 0) \wedge b \geq b^-).$$

Now, Rule MOD-S does apply.

Interface Automata

Introduced, studied, and extended in a recent burst of papers by de Alfaro, Henzinger, etc. A good starter:

> L. de Alfaro. Game Models for Open Systems. *Verification: Theory and Practice, LNCS 2772*, 269-289. Springer, 2003.

- A process language in the form of an automaton with joint actions (divided into inputs and outputs) for specifying the abstract behaviors of a module.
- Unreadiness to offer an input in a state is seen as assuming that the environment does not offer the corresponding output in the same state.
- So, one single interface automaton describes the input assumption and the output guarantee of a module.

Interface Automata (cont.)

- When two interface automata are composed, an *incompatible* state may result, where some output is enabled in one automaton but the corresponding input is not in the other automaton.
- Main decision problem: compatibility.
  Two interface automata are *compatible* if there exists an environment in which their product can be useful, i.e., all incompatible states may be avoided.
- It is possible to represent the assumption and the guarantee separately by a pair of I/O automata (which are input-enabled).
  This idea has recently been explored by K. Larsen *et al.* [FM 2006].

Concluding Remarks

- Assume-guarantee specification and reasoning were motivated by practical concerns.
- However, advancing the practice seems a lot harder than advancing the theory.
- It took over three decades for pre and post-conditions and state invariants to get gradually accepted in practice.
- Hopefully, more general assume-guarantee specifications will start to play a complementary role soon.