Compositional Specification and Reasoning

(Based on [Owicki and Gries 1976; Lamport 1980; Misra 1995; Jones 1983; Chandy and Misra 1981; Pnueli 1985; Abadi and Lamport 1995; Jonsson and Tsay 1996; de Alfaro 2003])

Yih-Kuen Tsay

Dept. of Information Management
National Taiwan University

Outline

- Review of the Owicki-Gries Method
- Compositional Methods
- The Mutual Induction Mechanism
- Compositional Reasoning in Temporal Logic
- Interface Automata
- Concluding Remarks

Sequential vs. Concurrent Programs/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:

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

- Proof outlines (for terminating programs)
- Interference freedom (here, one can sense the notion of "assume-guarantee")
- Auxiliary variables

Interference Freedom

- A proof outline $\{p_i\}\ S_i^*\ \{q_i\}$ *does not interfere* with another proof outline $\{p_j\}\ S_j^*\ \{q_j\}$ if the following holds:
  For every normal assignment or atomic region R in S_i and every assertion r in $\{p_j\}\ S_j^*\ \{q_j\}$,

$$\{r \wedge pre(R)\}\ R\ \{r\}.$$

- Given a parallel program $[S_1 \| \cdots \| S_n]$, the proof outlines $\{p_i\}\ S_i^*\ \{q_i\}$, $1 \leq i \leq n$, are said to be *interference free* if none of the proof outlines interferes with any other.

Main Composition Rule of Owicki and Gries

$$\frac{\{p_i\}\ S_i^*\ \{q_i\},\ 1 \le i \le n,\ \text{are interference free}}{\{\bigwedge_{i=1}^{n} p_i\}\ [S_1 \| \cdots \| S_n]\ \{\bigwedge_{i=1}^{n} q_i\}}$$

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, the purists would say, "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:

Lamport, L. 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:

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

- Rule of Parallel Composition:

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

UNITY Logic

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

Misra, J. 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:

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

- But, “*co*” is much less convenient for sequential composition.

Jones' Rely/Guarantee Pairs

Jones, C.B. 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 Specifications

- 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.
- The names vary: rely-guarantee, assumption-commitment, assumption-guarantee, etc.

Note: 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

Misra, J. and Chandy, K. 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:

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

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

Pnueli, A. 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:

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

- Main notation: $E \overset{+}{\rhd} M$
  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.

Abadi and Lamport

- Three kinds of implication (between safety properties A and G):
  - $A \rightarrow G$
    $\sigma \models A \rightarrow G \iff \sigma \models A$ implies $\sigma \models G$.
  - $A \mathrel{-\!\triangleright} G$
    $\sigma \models A \mathrel{-\!\triangleright} G \iff$ for all $i \geq 0$, $\sigma|_i \models A$ implies $\sigma|_i \models G$.
  - $A \overset{+}{\mathrel{-\!\triangleright}} G$
    $\sigma \models A \overset{+}{\mathrel{-\!\triangleright}} G \iff$ for all $i \geq 0$, $\sigma|_i \models A$ implies $\sigma|_{i+1} \models G$.
- Fundamental relationships
  - $A \overset{+}{\mathrel{-\!\triangleright}} G$ is the "realizable part" of $A \rightarrow G$.
  - $M \| A \models G$ iff $M \models A \mathrel{-\!\triangleright} G$.
  - $\models A \overset{+}{\mathrel{-\!\triangleright}} G = (G \mathrel{-\!\triangleright} A) \mathrel{-\!\triangleright} G$.
  - When A and G are "orthogonal", $\models A \overset{+}{\mathrel{-\!\triangleright}} G = A \mathrel{-\!\triangleright} G$ and hence $M \| A \models G$ iff $M \models A \overset{+}{\mathrel{-\!\triangleright}} G$.

Abadi and Lamport (cont.)

One of the composition rules:

$$\frac{\begin{array}{l}\models A \wedge G_2 \rightarrow A_1 \\ \models A \wedge G_1 \rightarrow A_2 \\ \models A \wedge G_1 \wedge G_2 \rightarrow G\end{array}}{\models (A_1 \overset{+}{\rightarrow\!\!\triangleright} G_1) \wedge (A_2 \overset{+}{\rightarrow\!\!\triangleright} G_2) \rightarrow (A \overset{+}{\rightarrow\!\!\triangleright} G)}$$

Alternative form:

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

Modular Reasoning in LTL

The operators $\rightarrowtriangle$ and $\overset{+}{\rightarrowtriangle}$ can be formalized in LTL:

Jonsson, B. and Tsay, Y.-K. 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.
- $\mathit{first} \triangleq \circleddash \mathit{false}$, 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

$$
\begin{array}{c}
\textbf{local } a, b : \textbf{integer where } a = b = 0 \\
P_a :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\; a := b + 1 \;\right] \end{array} \right] \parallel P_b :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\; b := a + 1 \;\right] \end{array} \right]
\end{array}
$$

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

Program keep-ahead(cont.)

$$\textbf{local } a, b : \textbf{integer where } a = b = 0$$

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

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

Modularized Program keep-ahead

```
module $M_a$
in $b$ : integer
out $a$ : integer  $= 0$
loop forever do
  $[\ a := b + 1\ ]$
```

$\|$

```
module $M_b$
in $a$ : integer
out $b$ : integer  $= 0$
loop forever do
  $[\ b := a + 1\ ]$
```

Modularized Program keep-ahead (cont.)

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

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

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:

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

or equivalently,

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

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

Refinement

- Refinement of Guarantee

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

- Refinement of Assumption

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

Composing A-G Specifications

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

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.

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

A Compositional Verification Rule

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

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

Interface Automata

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

> *de Alfaro, L. 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.

Concluding Remarks

- Assume-guarantee specification and reasoning were motivated by practical concerns.
- The effort had mostly been on obtaining the right form of specifications to enable compositional reasoning.
- 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.

References

- Abadi, M. and Lamport, L. Composing specifications. *TOPLAS*, 15(1):73–132, January, 1993.
- Abadi, M. and Lamport, L. Conjoining specifications. *TOPLAS*, 17(3):507–534, May, 1995.
- Abadi, M. and Plotkin, G.D. An abstract account of composition. *MFCS* 1995, 499–508.
- Abadi, M. and Merz, S. A logical view of composition. *TCS*, 114(1):3–30, June, 1993.
- de Alfaro, L. Game Models for Open Systems. *Verification: Theory and Practice, LNCS 2772*, 269-289. Springer, 2003.
- Apt, K.R. and Olderog, E.-R. *Verification of Sequential and Concurrent Programs*, Third Extended Edition, Springer-Verlag, 209.

References (cont.)

- Collette, P. An explanatory presentation of composition rules for assumption-commitment specifications. *IPL*, 50:31–35, 1994.
- Floyd, R.W. Assigning meanings to programs. *MACS*, 19–32, 1967.
- Hoare, C.A.R. An axiomatic basis for computer programs. *CACM*, 12(10):576–580, May, 1969.
- Jones, C.B. Tentative steps towards a development method for interfering programs. *TOPLAS*, 5(4):596-619, 1983.
- Jonsson, B. and Tsay, Y.-K. Assumption/guarantee specifications in linear-time temporal logic. *TCS*, 167:47–72, October, 1996.
- Lamport, L. The 'Hoare Logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

References (cont.)

- Misra, J. A logic for concurrent programming. *Journal of Computer and Software Engineering*, 3(2): 239-272, 1995.
- Misra, J. and Chandy, K. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.
- Owicki, S. and Gries, D. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
- Pnueli, A. In transition from global to modular temporal reasoning about programs. *Logics and Models of Concurrent Systems*, 123-144. Springer, 1985.