Compositional Reasoning Methods

Alma Juarez
`aljuarez@cs.uwaterloo.ca`
DC 2551D, x7867

WATFORM

ECE725/CS745: Computer-Aided Verification
University of Waterloo

March, 2006

Outline

Motivation

Verification

Complex systems, and especially safety-critical ones are in need of formal verification.

- Which style of proof method is more appropriate?

Motivation

Verification

Recall (from lecture 2):

Verification involves checking a satisfaction relation, usually in the form of a sequent:

$$\mathcal{M} \models \phi$$

where

- $\mathcal{M}$ is a model (or implementation)
- ϕ is a property (or specification)
- $\models$ is a relationship that should hold between $\mathcal{M}$ and ϕ, i.e., $(\mathcal{M},\phi) \in \models$

We say that the model satisfies or “has” the property, or that we can conclude the property from the model.

Motivation

Verification

Recall (from lecture 2):

Verification involves:

1. specifying the model/system/implementation
   Modelling language
2. specifying the property/specification
   Logic
3. choosing the satisfaction relation
   Formula in logic
4. checking the satisfaction relation
   Verification engine

These 4 steps are NOT independent.

Motivation
Verification

Recall (from lecture 2):

Verification involves:

1. specifying the model/system/implementation
   $\longrightarrow$ Modelling language
2. specifying the property/specification
   $\longrightarrow$ Logic
3. choosing the satisfaction relation
   $\longrightarrow$ Formula in logic
4. checking the satisfaction relation
   $\longrightarrow$ Verification engine

These 4 steps are NOT independent.

Motivation

Logic and Verification

Recall (from lecture 2):

Different logics give us different ways of expressing $\mathcal{M}$ and ϕ define the pairs that are members of $\models$.

Hopefully the calculation of the satisfaction relation is compositional in either the property or the model. This decomposes the verification task.

The model and property both describes sets of “behaviours”. The satisfaction relation is a relation between the set of behaviours of the model and the set of behaviours of the property.

Alternatives

Decomposition / Composition

Composition

Two models may be used to define another more complex model.

Decomposition

When a model is split into smaller, less complex models.

Alternatives

Decomposition / Composition

Composition

Two models may be used to define another more complex model.

Decomposition

When a model is split into smaller, less complex models.

Depends on

- Modelling language
- Logic
- Verification engine

Alternatives

Verification alternatives

Modelling language

- Finite state machines
- Labelled transition systems
- Petri nets
- Timed automata
- Process algebra
- Operational semantics
- Denotational semantics
- Hoare's logic

Alternatives

Verification alternatives

Logic

- Propositional logic
- Predicate logic
- Higher order logic
- Linear temporal logic (LTL)
- Computational tree logic (CTL).

Alternatives

Verification alternatives

Verification engine

- Theorem proving
- Model checking

Alternatives
Verification alternatives

Recall (from lecture 2):

Verification involves:

1. specifying the model/system/implementation
   $\longrightarrow$ Modelling language
2. specifying the property/specification
   $\longrightarrow$ Logic
3. choosing the satisfaction relation
   $\longrightarrow$ Formula in logic
4. checking the satisfaction relation
   $\longrightarrow$ Verification engine

These 4 steps are NOT independent.

Alternatives

Model / System / Implementation

- **Sequentiality:** property of systems which consist of a computation that execute in order and consecutively without interruptions. (Program that runs on a single processor and has all the resources available)

Alternatives

Model / System / Implementation

- **Sequentiality:** property of systems which consist of a computation that execute in order and consecutively without interruptions. (Program that runs on a single processor and has all the resources available)
- **Concurrency:** property of systems which consist of computations that execute overlapped in time, and which may permit the sharing of common resources between those overlapped computations. (Program that consists of a collection of processes and shared objects, such as shared channels and/or shared variables)

Alternatives

Sequentiality / Concurrency Characterization

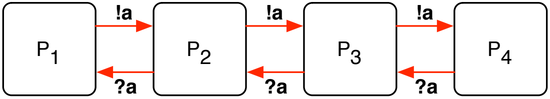
- **Sequential program:** sufficient to observe their pairs of initial and corresponding final states (*observable behaviour*). Two different sequential programs having the same observational behaviour are regarded as equivalent, so from this point of view they are "atomic" units.

(a=2, b=0)	**(a=2, b=0)**
{ b=a*2; b=b-1;	{ b=a*3; b=b/2;
(a=2, b=3)	**(a=2, b=3)**

Alternatives

Sequentiality / Concurrency Characterization

- **Concurrent program:** due to the possibility of synchronization and communication between such programs, intermediate states are as important as final ones. Hence, the observational behaviour should include values of those variables shared between the processes or the messages communicated between them.

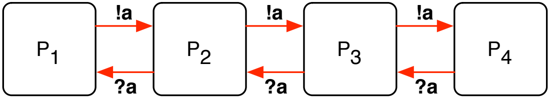


(a > 0)

Alternatives

Sequentiality / Concurrency Characterization

- **Concurrent program:** due to the possibility of synchronization and communication between such programs, intermediate states are as important as final ones. Hence, the observational behaviour should include values of those variables shared between the processes or the messages communicated between them.



(a > 0)

Alternatives
Synchronization

Synchronization allows coordination with respect to time so meaningful communication between processes can occur.

- **Mutual exclusion:** groups actions into *critical sections* that are never interleaved during execution. (transactions)
- **Conditional synchronization:** delays a process until the state satisfies some specified condition. (locks)
- **Synchronous communication:** gives the impression that communication between processes is simultaneous. (Server and receiver must be running)
- **Asynchronous communication:** the communication can be delayed. (Intermediate buffer or queue holds the message)

Alternatives

Communication

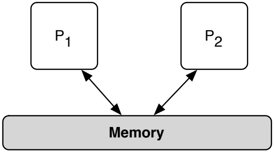
Communication allows one process to influence execution of another one and can be accomplished using:

- **Shared variables:** External processes have access to a pool of shared memory cells.
- **Message passing:** Every process has its local memory, and processes share channels.

Alternatives

Communication

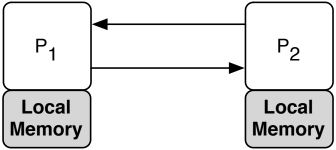
- **Shared variables:** External processes have access to a pool of shared memory cells. What should be prevented is that a process is able to access information in this shared memory while the information is changed by another process, and vice versa, since this would cause that information become temporarily inconsistent.



Alternatives

Communication

- **Message passing:** Every process has its local memory, and processes share channels. What we need to guarantee for message passing to work is that when a message is sent from one process along a channel, the next process in the chain receives the message and eventually reacts to it.



Alternatives

Types of concurrency

Recall (from lecture 5):

Maximum Parallelism (synchronous): All assignments are executed simultaneously, i.e., all modules perform all of their atomic assignments at the same time. This is the default in SMV.

Interleaving Parallelism (asynchronous): Module executions are interleaved. Each module performs all of its atomic assignments in isolation. Multiple modules do not execute in the same step. In the modules that aren't executing in a step, the variables do not change their values.

Alternatives

Types of processes/components

- **Homogeneous:** All components are alike, performing the same kind of computations.
- **Heterogeneous:** Components are unlike, performing different kinds of functions or computations.

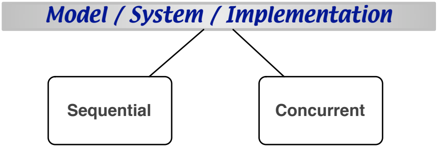
Alternatives

Number of processes/components

- **Bounded:** We know the total number of components in advance (related to static creation of processes).
- **Unbounded:** We do not know the total number of components in advance (related to dynamic creation of processes).

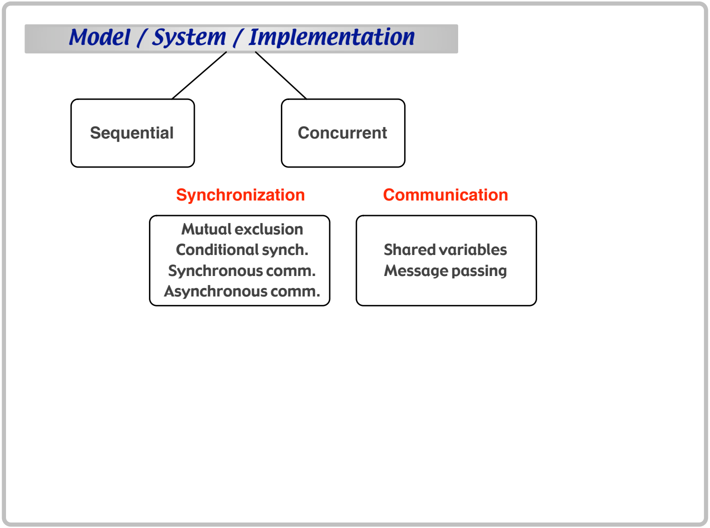
Alternatives

Model / System / Implementation (Summary)



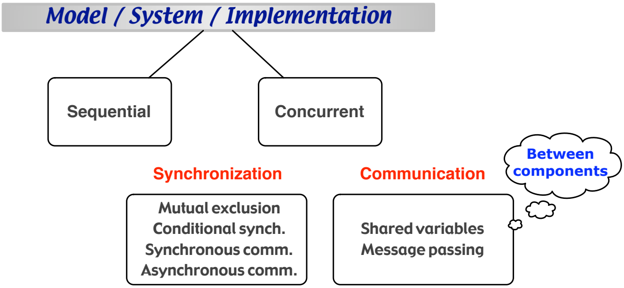
Alternatives

Model / System / Implementation (Summary)



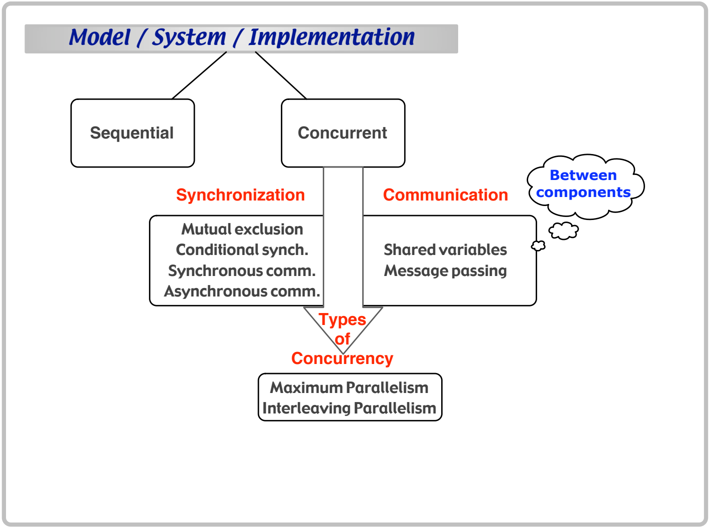
Alternatives

Model / System / Implementation (Summary)



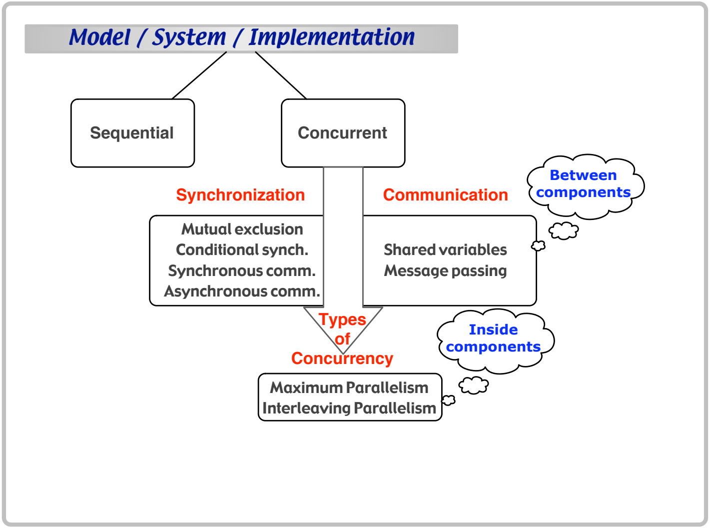
Alternatives

Model / System / Implementation (Summary)



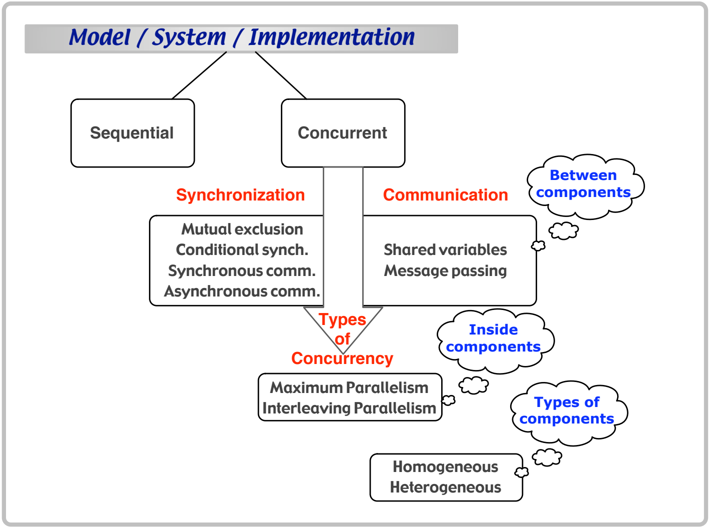
Alternatives

Model / System / Implementation (Summary)



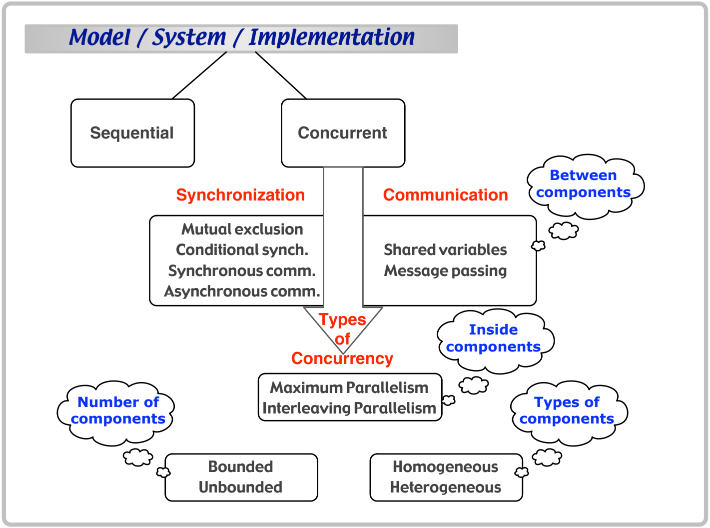
Alternatives

Model / System / Implementation (Summary)



Alternatives

Model / System / Implementation (Summary)



Methods
Assume-Guarantee

Assume-Guarantee: This technique verifies each component process separately.

- Suppose that there are two processes $\mathcal{M}$ and $\mathcal{M}'$.
- Behaviour of process $\mathcal{M}$ depends on the behaviour of process $\mathcal{M}'$
  $\longrightarrow$ user specifies a set of assumptions that must be satisfied by $\mathcal{M}'$ in order to guarantee the correctness of process $\mathcal{M}$.
  $\longrightarrow$ vice versa.

Methods

Assume-Guarantee

Assume-Guarantee: This technique verifies each component process separately.

- Typically, a formula is a triple $< g > \mathcal{M} < f >$, whenever $\mathcal{M}$ is part of a system satisfying the assumption g, the system must also guarantee the property f.
- The proof strategy, expressed as an inference rule:

$$\frac{\begin{array}{c} < \mathit{true} > \mathcal{M} < g > \\ < g > \mathcal{M}' < f > \end{array}}{< \mathit{true} > \mathcal{M} \| \mathcal{M}' < f >}$$

Methods
Assume-Guarantee

Assume-Guarantee: This technique verifies each component process separately.

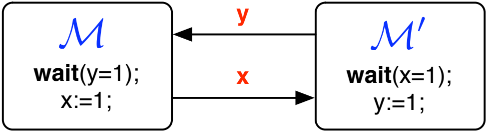
- It is important to avoid circularity in assume-guarantee proofs!
- The following rule is unsound:

$$\frac{\begin{array}{c} < g > \mathcal{M} < f > \\ < f > \mathcal{M}' < g > \end{array}}{\mathcal{M} \| \mathcal{M}' \not\models < f \wedge g >}$$

Methods

Assume-Guarantee

Example on assume-guarantee circularity:



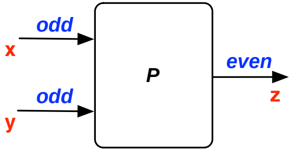
$$\frac{\begin{array}{c} < AF(y=1) > (\mathcal{M} : \textbf{wait}(y=1); x := 1;) < AF(x=1) > \\ < AF(x=1) > (\mathcal{M}' : \textbf{wait}(x=1); y := 1;) < AF(y=1) > \end{array}}{\mathcal{M} \| \mathcal{M}' \not\models < AF(x=1) \wedge AF(y=1) >}$$

From *Clark, et.al.* "Model Checking"

Methods

Assume-Guarantee Example

Consider an adder component P that adds two input numbers x and y, and places the output in z. (Natarajan Shankar)



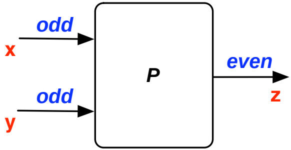
Here x and y, and z can be program variables, signals, or latches, depending on the chosen model of computation.

From *de Rover, et.al.* "Concurrency Verification"

Methods

Assume-Guarantee Example

Consider an adder component P that adds two input numbers x and y, and places the output in z. (Natarajan Shankar)



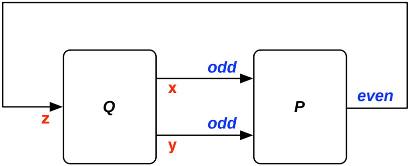
Here x and y, and z can be program variables, signals, or latches, depending on the chosen model of computation.

P by itself cannot unconditionally guarantee the property that the output of z to be an even number!!!

Methods

Assume-Guarantee Example

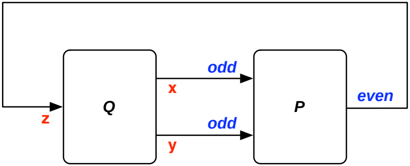
Now P is composed with another component Q that generates the inputs x and y.



Methods

Assume-Guarantee Example

Now P is composed with another component Q that generates the inputs x and y.

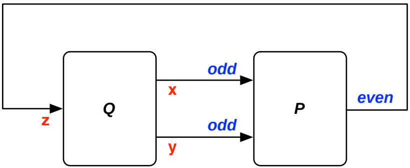


Circularity!!!

Methods

Assume-Guarantee Example

Now P is composed with another component Q that generates the inputs x and y.



The circularity can be broken:
*The **guarantee** that output z is even is satisfied by P as long as the **assumption** that its previous inputs x and y are odd has always been satisfied before (temporal induction) by P's environment.*

Methods

Assume-Guarantee variants

Main variants of the A-G paradigm:

- **Assumption-Commitment:** Variant for **message passing** systems, discovered by Jayadev Misra and Mani Chandy in 1981.
- **Rely-Guarantee:** Variant for **shared-variable** concurrency, discovered by Cliff Jones in 1981/1983.

Methods
Assumption-Commitment

Assumption-Commitment (A-C) $\rightarrow$ message passing systems
Formally, an A-C formula has the form:

$$<A, C> : \{ \varphi \} \, P \, \{ \Psi \}$$

where

- P denotes a program
- A, φ, Ψ, C denote predicates.

Methods
Assumption-Commitment

Assumption-Commitment (A-C) $\rightarrow$ message passing systems
Formally, an A-C formula has the form:

$$<A, C> : \{ \varphi \} \, P \, \{ \Psi \}$$

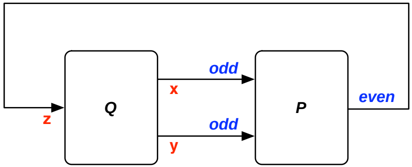
Informally:
If φ holds in the initial state, including the communication history, in which P starts its execution, then

- *C holds initially, and C holds after every communication provided A holds after all preceding communications, and*
- *if P terminates and A holds after all previous communications (including the last one) then Ψ holds in the final state including the final communication history.*

Methods

Assumption-Commitment Example

P is composed with another component Q that generates the inputs x and y.



Reasoning using the Assumption-Commitment paradigm

Methods

Assumption-Commitment Example

The proof strategy, expressed as an inference rule:

$$\frac{\begin{array}{c} < \mathit{true} > \mathcal{M} < g > \\ < g > \mathcal{M}' < f > \end{array}}{< \mathit{true} > \mathcal{M} \| \mathcal{M}' < f >}$$

Methods

Assumption-Commitment Example

The proof strategy, expressed as an inference rule:

$$\frac{\begin{array}{c} < \mathit{true} > \mathcal{M} < g > \\ < g > \mathcal{M}' < f > \end{array}}{< \mathit{true} > \mathcal{M} \| \mathcal{M}' < f >}$$

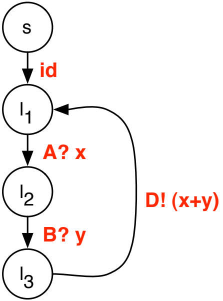
Inference rule in terms of A-C formulas:

$$\frac{\begin{array}{c} < A_1, C_1 >: \{\varphi_1\} P \{\Psi_1\} \\ < A_2, C_2 >: \{\varphi_2\} Q \{\Psi_2\} \end{array}}{< A_1, C_2 >: \{\varphi_1 \wedge \varphi_2\} P \| Q \{\Psi_1 \wedge \Psi_2\}}$$

Methods

Assumption-Commitment Example

An implementation of P using message passing:



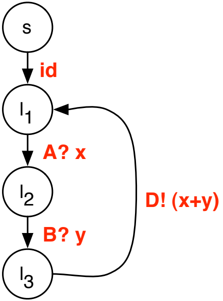
#chan – number of communications via *chan*

last(chan) – latest value sent via *chan*

Methods

Assumption-Commitment Example

An implementation of *P* using message passing:



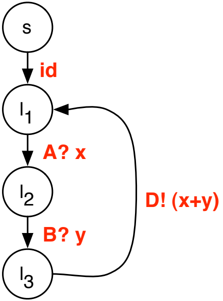
A-C correctness formula for *P* in an arbitrary environment:

$$<true, \#D = \#A = \#B \geq 1 \rightarrow \\ last(D) = last(A) + last(B) > :$$

$$\{ \#D = \#A = \#B = 0 \}\ P\ \{ false \}$$

Methods
Assumption-Commitment Example

An implementation of *P* using message passing:



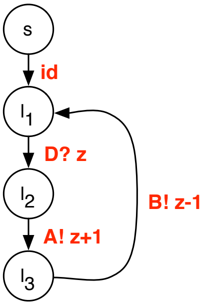
We obtain for *P* the following A-C correctness formula:

$$<(\#A \geq 1 \rightarrow odd(last(A))) \wedge (\#B \geq 1 \rightarrow odd(last(B))),$$
$$(\#A \geq 1) \wedge (\#B \geq 1) \wedge (\#D \geq 1) \rightarrow even(last(D)) >:$$
$$\{\ \#D = \#A = \#B = 0\ \}\ P\ \{\ false\ \}$$

Methods

Assumption-Commitment Example

An implementation of Q using message passing:



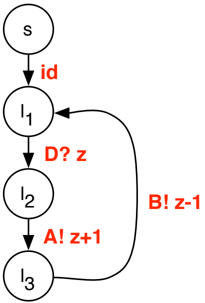
A-C correctness formula for Q in an arbitrary environment:

$$<true, \#D = \#A = \#B \geq 1 \rightarrow (last(A) = last(D) + 1) \wedge (last(B) = last(D) - 1) > : \{\ \#D = \#A = \#B = 0\ \}\ Q\ \{\ false\ \}$$

Methods

Assumption-Commitment Example

An implementation of Q using message passing:



We obtain for Q the following A-C correctness formula:

$$<(\#A \geq 1) \wedge (\#B \geq 1) \wedge (\#D \geq 1) \rightarrow even(last(D)),$$
$$(\#A \geq 1) \wedge (\#B \geq 1) \wedge (\#D \geq 1) \rightarrow odd(last(A))) \wedge odd(last(B)) >:$$
$$\{ \#D = \#A = \#B = 0 \} \; Q \; \{ \mathit{false} \}$$

Methods

Assumption-Commitment Example

For the parallel composition:

$<A_1, C_1>$: $\{\ \varphi_1\ \}\ P\ \{\ \Psi_1\ \}$

$$<(\#A \geq 1 \rightarrow odd(last(A))) \wedge (\#B \geq 1 \rightarrow odd(last(B))),$$
$$(\#A \geq 1) \wedge (\#B \geq 1) \wedge (\#D \geq 1) \rightarrow even(last(D)) > :$$
$$\{\ \#D = \#A = \#B = 0\ \}\ P\ \{\ false\ \}$$

$<A_2, C_2>$: $\{\ \varphi_2\ \}\ Q\ \{\ \Psi_2\ \}$

$$<(\#A \geq 1) \wedge (\#B \geq 1) \wedge (\#D \geq 1) \rightarrow even(last(D)),$$
$$(\#A \geq 1) \wedge (\#B \geq 1) \wedge (\#D \geq 1) \rightarrow odd(last(A))) \wedge odd(last(B)) >:$$
$$\{\ \#D = \#A = \#B = 0\ \}\ Q\ \{\ false\ \}$$

$<A_1, C_2>$: $\{\ \varphi_1 \wedge \varphi_2\ \}\ P \| Q\ \{\ \Psi_1 \wedge \Psi_2\ \}$

$$<(\#A \geq 1 \rightarrow odd(last(A))) \wedge (\#B \geq 1 \rightarrow odd(last(B))),$$
$$(\#A \geq 1) \wedge (\#B \geq 1) \wedge (\#D \geq 1) \rightarrow odd(last(A))) \wedge odd(last(B)) >:$$
$$\{\ \#D = \#A = \#B = 0\ \}\ P \| Q\ \{\ false\ \}$$

Methods

Rely-Guarantee

Formally an R-G formula has the form:

$$<\mathit{rely}, \mathit{guar}> : \{ \varphi \} \mathbf{P} \{ \Psi \}$$

Traditionally,

- φ and Ψ impose conditions upon the initial and final state of the computation of P
- *rely* and *guar* impose conditions upon environmental transitions and transitions on P itself.

Methods
Rely-Guarantee

Formally an R-G formula has the form:

$$<\mathit{rely}, \mathit{guar}> : \{ \varphi \} \; \mathbf{P} \; \{ \Psi \}$$

If,

- *P is invoked in an initial state which satisfies* φ*, and*
- *whenever at some moment during the computation of P all past environmental transitions satisfy rely ,*

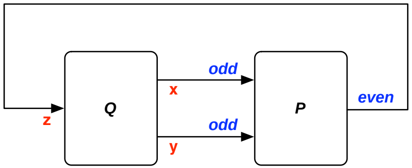
then,

- *all transitions by P up to that moment satisfy guard , and*
- *if this computation terminates, its final state satisfy* Ψ.

Methods

Rely-Guarantee Example

P is composed with another component Q that generates the inputs x and y.



Reasoning using the Rely-Guarantee paradigm

Methods

Rely-Guarantee Example

The proof strategy, expressed as an inference rule:

$$\frac{\begin{array}{c} < \textit{true} > \mathcal{M} < g > \\ < g > \mathcal{M}' < f > \end{array}}{< \textit{true} > \mathcal{M} \| \mathcal{M}' < f >}$$

Methods

Rely-Guarantee Example

The proof strategy, expressed as an inference rule:

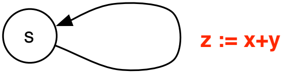
$$\frac{\begin{array}{c} < \mathit{true} > \mathcal{M} < g > \\ < g > \mathcal{M}' < f > \end{array}}{< \mathit{true} > \mathcal{M} \| \mathcal{M}' < f >}$$

Inference rule in terms of R-G formulas:

$$\frac{\begin{array}{c} < R_1, G_1 >: \{\varphi_1\} P \{\Psi_1\} \\ < R_2, G_2 >: \{\varphi_2\} Q \{\Psi_2\} \end{array}}{< R_1, G_2 >: \{\varphi_1 \wedge \varphi_2\} P \| Q \{\Psi_1 \wedge \Psi_2\}}$$

Methods
Rely-Guarantee Example

An implementation of P using shared-variable:



σ' – action produced by σ.

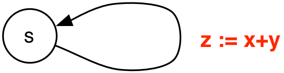
R-G correctness formula for P in an arbitrary environment:

<*true*, z' = x+y > : { *true* } *P* { *false* }

Methods

Rely-Guarantee Example

An implementation of P using shared-variable:



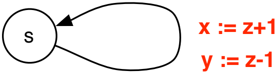
We obtain for P the following A-C correctness formula:

$$<z = z' \wedge odd(x') \wedge odd(y'), even(z') > :$$
$$\{\ odd(x) \wedge odd(y)\ \}\ P\ \{\ false\ \}$$

Methods

Rely-Guarantee Example

An implementation of Q using shared-variable:



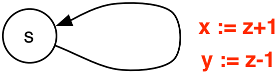
R-G correctness formula for Q in an arbitrary environment:

$$\langle \mathit{true},\ x' = z+1 \wedge y' = z-1 \rangle :$$
$$\{\ \mathit{true}\ \}\ P\ \{\ \mathit{false}\ \}$$

Methods

Rely-Guarantee Example

An implementation of Q using shared-variable:



We obtain for Q the following A-C correctness formula:

$$< x = x' \wedge y = y' \wedge \mathit{even}(z'), \mathit{odd}(x') \wedge \mathit{odd}(y') > :$$
$$\{ \mathit{even}(z) \} \; Q \; \{ \mathit{false} \}$$

Methods

Rely-Guarantee Example

For the parallel composition:

$\langle R_1, G_1 \rangle$: $\{ \varphi_1 \}\ P\ \{ \Psi_1 \}$

$$\langle z = z' \wedge \mathit{odd}(x') \wedge \mathit{odd}(y'), \mathit{even}(z') \rangle :$$
$$\{ \mathit{odd}(x) \wedge \mathit{odd}(y) \}\ P\ \{ \mathit{false} \}$$

$\langle R_2, G_2 \rangle$: $\{ \varphi_2 \}\ Q\ \{ \Psi_2 \}$

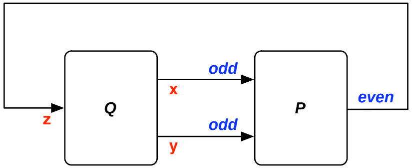
$$\langle x = x' \wedge y = y' \wedge \mathit{even}(z'), \mathit{odd}(x') \wedge \mathit{odd}(y') \rangle :$$
$$\{ \mathit{even}(z) \}\ Q\ \{ \mathit{false} \}$$

$\langle R_1, G_2 \rangle$: $\{ \varphi_1 \wedge \varphi_2 \}\ P \| Q\ \{ \Psi_1 \wedge \Psi_2 \}$

$$\langle z = z' \wedge \mathit{odd}(x') \wedge \mathit{odd}(y'),$$
$$\mathit{odd}(x') \wedge \mathit{odd}(y') \wedge z = z' \rangle :$$
$$\{ \mathit{odd}(x) \wedge \mathit{odd}(y) \wedge \mathit{even}(z) \}\ P \| Q\ \{ \mathit{false} \}$$

Methods

Rely-Guarantee Example



The extra element for the network to function is:

- guarantee-part of *Q*'s specification should imply the rely-part of *P*'s specification.
- guarantee-part of *P*'s specification should imply the rely-part of *Q*'s specification.

Methods

Justifying Assume-Guarantee Proofs

Recall (from lecture 5):

Definition of Simulation

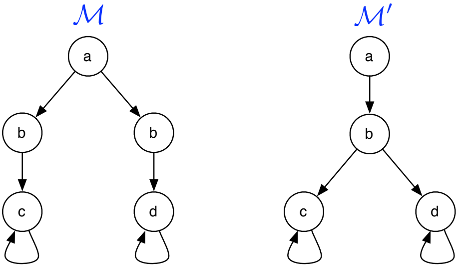
Two Kripke structures,
$\mathcal{M} = (AP, S, R, S_0, L)$ and
$\mathcal{M}' = (AP', S', R', S_0', L')$, where $AP' \subseteq AP$,
$\mathcal{M} \preceq \mathcal{M}'$ if there exists a $H \subseteq S \times S'$ such that:

- initial states:
  - $\forall s_0 \in S_0 \cdot \exists s_0' \in S_0' \cdot H(s_0, s_0')$
- steps: $\forall s, s' \cdot$ if $H(s, s')$, then
  - $L(s) \bigcap AP' = L(s')$
  - $\forall s_1 \cdot R(s, s_1) \Rightarrow \exists s_1' \cdot R'(s', s_1') \wedge H(s_1, s_1')$

H is called a simulation relation. We say $\mathcal{M}'$ simulates $\mathcal{M}$.

Methods
Justifying Assume-Guarantee Proofs

Simulation



Methods

Justifying Assume-Guarantee Proofs

Lem33: $\preceq$ is a preorder on the set of structures.

Thm16: Suppose $\mathcal{M} \preceq \mathcal{M}'$. Then for every ACTL* formula f (with atomic propositions in AP'):

$$\mathcal{M}' \models f \text{ implies } \mathcal{M} \models f$$

- $\preceq_F$ is a preorder on fair structures.

Thm17: If $\mathcal{M} \preceq_F \mathcal{M}'$, then for every ACTL* formula f interpreted over fair paths, $\mathcal{M}' \models_F f$ implies $\mathcal{M} \models_F f$.

From *Clark, et.al.* "Model Checking"

Methods

Justifying Assume-Guarantee Proofs

- For any ACTL* formula f it is possible to construct a special model $\mathcal{T}_f$, called **tableau** .

Thm18: the tableau $\mathcal{T}_f$ has the property that:

$$\mathcal{M} \models_F f \text{ iff } \mathcal{M}' \preceq_F \mathcal{T}_f$$

Methods

Justifying Assume-Guarantee Proofs

Now, the first inference:

$$\frac{\begin{array}{c} <\mathit{true}>\mathcal{M}<g> \\ <g>\mathcal{M}'<f> \end{array}}{<\mathit{true}>\mathcal{M}\|\mathcal{M}'<f>}$$

can be rewritten using the logic fair ACTL* and $\preceq_F$

$$\frac{\begin{array}{c} \mathcal{M} \preceq_F \mathcal{T}_g \\ \mathcal{M}'\|\mathcal{T}_g \models_F f \end{array}}{\mathcal{M}\|\mathcal{M}' \models_F f}$$

Methods

Justifying Assume-Guarantee Proofs

Thm19: For all $\mathcal{M}$ and $\mathcal{M}'$, $\mathcal{M} \parallel \mathcal{M}' \preceq_F \mathcal{M}$.

Thm20: For all $\mathcal{M}$ and $\mathcal{M}'$,
If $\mathcal{M} \preceq_F \mathcal{M}'$ then $\mathcal{M} \parallel \mathcal{M}'' \preceq_F \mathcal{M}' \parallel \mathcal{M}''$.

Thm21: For all $\mathcal{M}$, $\mathcal{M} \preceq_F \mathcal{M} \parallel \mathcal{M}$.

Methods

Justifying Assume-Guarantee Proofs

To justify the A-G rule:

$$\frac{\begin{array}{r} < \mathit{true} > \mathcal{M} < \mathcal{A} > \\ < \mathcal{A} > \mathcal{M}' < g > \\ < g > \mathcal{M} < f > \end{array}}{< \mathit{true} > \mathcal{M} \| \mathcal{M}' < f >}$$

where $\mathcal{A}$, $\mathcal{M}$, $\mathcal{M}'$ represent finite state models and g and f represent fair ACTL* formulas. This corresponds to the proof rule:

$$\frac{\begin{array}{r} \mathcal{M} \preceq_F \mathcal{A} \\ \mathcal{A} \| \mathcal{M}' \preceq_F g \\ \mathcal{T}_g \| \mathcal{M} \models_F f \end{array}}{\mathcal{M} \| \mathcal{M}' \models_F f}$$

Methods

Justifying Assume-Guarantee Proofs

1.	$\mathcal{M} \preceq_F \mathcal{A}$	HYP1
2.	$\mathcal{M} \| \mathcal{M}' \preceq_F \mathcal{A} \| \mathcal{M}'$	L1, **Thm 20**
3.	$\mathcal{A} \| \mathcal{M}' \models_F g$	HYP2
4.	$\mathcal{A} \| \mathcal{M}' \preceq_F \mathcal{T}_g$	L3, **Thm 18**
5.	$\mathcal{M} \| \mathcal{M}' \preceq_F \mathcal{T}_g$	L2, L4, Tr $\preceq_F$
6.	$\mathcal{M} \| \mathcal{M} \| \mathcal{M}' \preceq_F \mathcal{T}_g \| f$	L5, **Thm 20**
7.	$\mathcal{T}_g \| f \preceq_F f$	HYP3
8.	$\mathcal{M} \| \mathcal{M} \| \mathcal{M}' \preceq_F f$	L6, L7, **Thm 17**
9.	$\mathcal{M} \preceq_F \mathcal{M} \| \mathcal{M}$	**Thm 21**
10.	$\mathcal{M} \| \mathcal{M}' \preceq_F \mathcal{M} \| \mathcal{M} \| \mathcal{M}'$	L9, **Thm 20**
11.	$\mathcal{M} \| \mathcal{M}' \preceq_F f$	L8, L10, **Thm 17**

Methods

Finite/Infinite State Systems

The theory of *computability* shows that there cannot be an algorithms that decides whether an arbitrary program terminates.

$\longrightarrow$ Most proof systems cannot be completelly automated.

Methods

Finite/Infinite State Systems

Deductive verification: can be used for reasoning about systems with infinitely many reachable states and can handle unrestricted programs with rich data-structures.
Technique: not fully automatic, user interaction to guide a theorem proving tool.

Model checking: technique for verifying finite state concurrent systems, and verification can be performed automatically by using an exhaustive search of the state space of the system.
Technique: performed automatically, preferable to deductive verification.

Methods

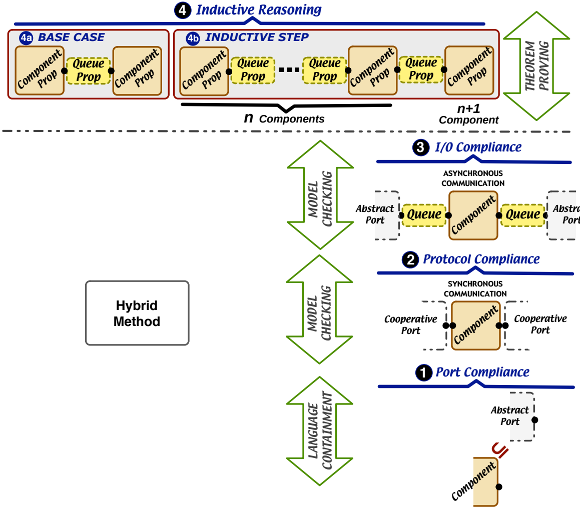
Finite/Infinite State Systems

Challenges: unbounded message queues, bounded but unknown number of components, dynamic process creation, unlike components.

New method: Hybrid verification – integrate deductive verification and model checking, so that the finite state parts can be verified automatically.

Methods

Finite/Infinite State Systems



From *Juarez*, "Verification of DFC Call Protocol Correctness Criteria"

Summary

- Compositional methods enable reasoning about complex systems by reducing their properties to the properties of their components.
- Key idea: *If*
  - we can deduce that the system satisfies each local property,
  - we know that the conjunction of the local properties implies the overall specification,

  then we can conclude that the complete system satisfy this specification as well.

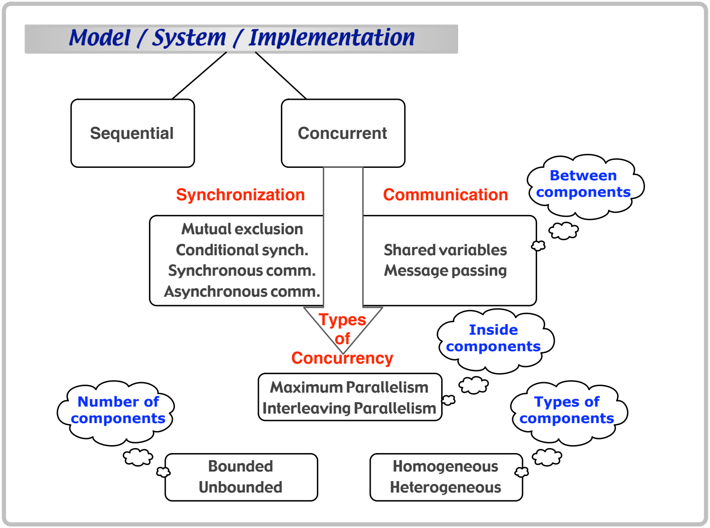
Summary

To remember:

- Compositional methods enable reasoning about complex systems by reducing their properties to the properties of their components.
- Verification involves 4 steps, related to:
  Modelling language, Logic, Formula in logic, Verification engine.
- The calculation of the satisfaction relation could be compositional in either the property or the model, decomposing the verification tasks.
- Main methods: Based on Assume-Guarantee paradigm, Hybrid.

Summary

To remember: (Alternatives)



Contact Information I

Alma L. Juarez Dominguez

Office: DC 2551D
Extension: x7867
e-mail: `aljuarez@cs.uwaterloo.ca`
Webpage: `www.cs.uwaterloo.ca/ aljuarez`

Comments are welcome!

References I

- E. M. Clarke, O. Grumberg, D. A. Peled.
  Model Checking. The MIT Press, 2002
- W. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel.
  Concurrency Verification. Cambridge University Press, 2001
- A. L. Juarez Dominguez.
  Verification of the DFC Call Protocol Correctness Criteria. Master's Thesis, 2005