Comparing models. Abstraction. Compositional reasoning

28 octombrie 2003

Problem setting

Specification formulas can be converted to automata
(LTL tableau construction)
– represent "simplest" system that conforms to the specification

When using an automaton as specification:
– what does it mean to say "system functions like this automaton"

How does one build (abstract) a simpler model from a complex one ?
Does verifying a simpler model ensure correctness of the initial one ?

Can one deduce correctness of a composite model from proving properties of the components ?

Language inclusion (trace inclusion)

Consider a Kripke structure M with a set AP of atomic propositions

Language of M = set of execution traces seen as sequences of labels
Formally: $\mathcal{L}(M)$ = set of infinite words (strings) $\alpha_0\alpha_1\alpha_2\ldots$
such that there exists a path $s_0s_1s_2\ldots$ of M with $L(s_i) = \alpha_i$.

Language inclusion preserves LTL properties:

$$\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S}) \Leftrightarrow \forall \mathbf{A}f \in LTL \,.\, \mathcal{S} \models \mathbf{A}f \Rightarrow \mathcal{M} \models \mathbf{A}f$$

Simulation relation

Consider two structures M and M', with $AP \supseteq AP'$. A relation $\preceq \subseteq S \times S'$ is a *simulation* relation between M and M' iff $\forall s \preceq s'$:
– $L(s) \cap AP' = L'(s')$ (s and s' labeled identically with respect to AP')
– $\forall s_1$ with $s \to s_1$ there exists s'_1 with $s' \to s'_1$ and $s_1 \preceq s'_1$
(any successor of s is simulated by a successor of s')

The structure M' simulates M ($M \preceq M'$) of there exists a simulation relation $\preceq$ such that for the initial states: $\forall s_0 \in S_0 \, \exists s'_0 \in S'_0 \, . \, s_0 \preceq s'_0$

Prop.: The simulation relation is a *preorder* over the set of structures (reflexive and transitive). We choose: $s \preceq s'' \Leftrightarrow \exists s' \, . \, s \preceq_1 s' \wedge s' \preceq_2 s''$

Theorem: If $M \preceq M'$, then $M' \models f \Rightarrow M \models f$, for any ACTL* formula f over AP'.

Bisimulation relation

Let M and M' be two structures with $AP' = AP$. A relation $\simeq \subseteq S \times S'$ is a *bisimulation* relation between M and M' iff $\forall s, s'$ with $s \simeq s'$:
– L(s) = L(s')
– $\forall s_1$ with $s \rightarrow s_1$ there exists s'_1 with $s' \rightarrow s'_1$ and $s_1 \simeq s'_1$
– $\forall s'_1$ with $s' \rightarrow s'_1$ there exists s_1 with $s \rightarrow s_1$ and $s_1 \simeq s'_1$
(or: $\simeq$ a *symmetric* simulation relation between M and M' *and* between M' and M)

Structures M and M' are *bisimilar* if there exists a bisimulation relation $\simeq$ such that for initial states: $\forall s_0 \in S_0\ \exists s'_0 \in S'_0\ .\ s_0 \simeq s'_0$, and $\forall s'_0 \in S'_0\ \exists s_0 \in S_0\ .\ s_0 \simeq s'_0$.

Prop. : The bisimulation relation is an equivalence relation among structures

Theorem: If $M \simeq M'$ then $\forall f \in CTL*$, $M \models f \Leftrightarrow M' \models f$.
Conversely: Two structures that satisfy the same CTL* (or even CTL) formulas are bisimilar (equivalently: two structures which are not bisimilar can be distinguished by a CTL formula).

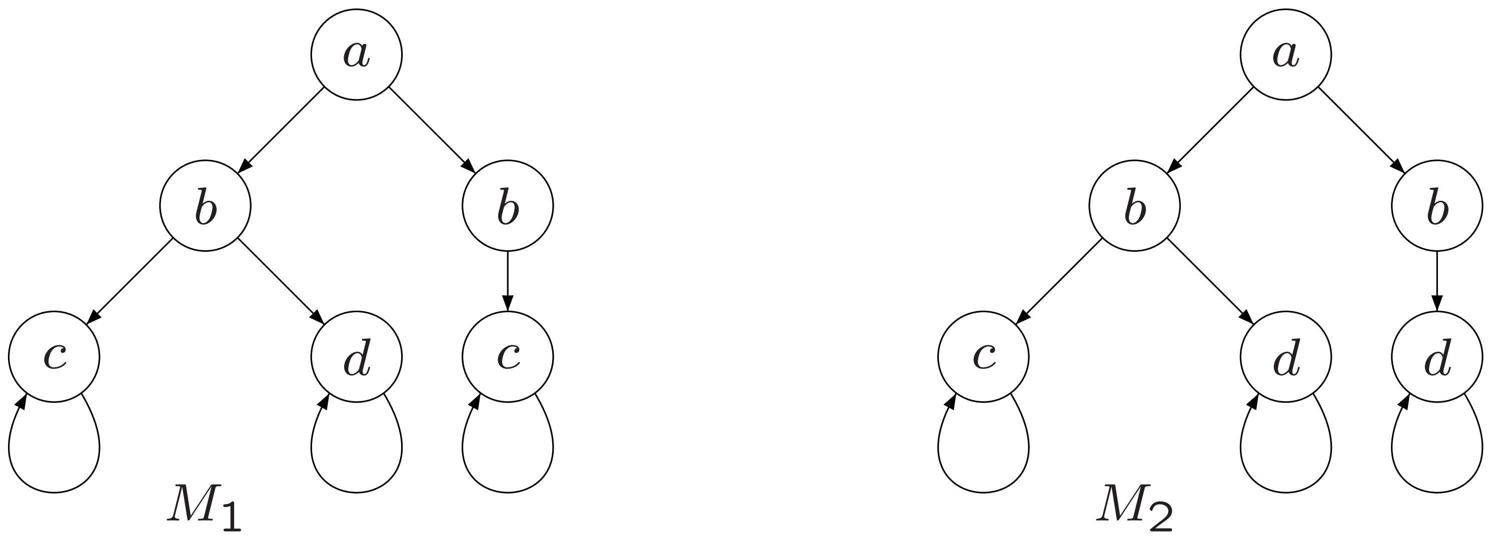
Example: language inclusion and simulation

Generally: $M \preceq M' \Rightarrow \mathcal{L}(M)|_{AP'} \subseteq \mathcal{L}(M')$

In the figure: $\mathcal{L}(M_1) = \mathcal{L}(M_2)$, $M_1 \preceq M_2$, $M_2 \not\preceq M_1$

Equivalent definition (game theory): $M \preceq M'$ if any move in M can be matchd by an equally labelled move in M'.

Example: simulation and bisimulation



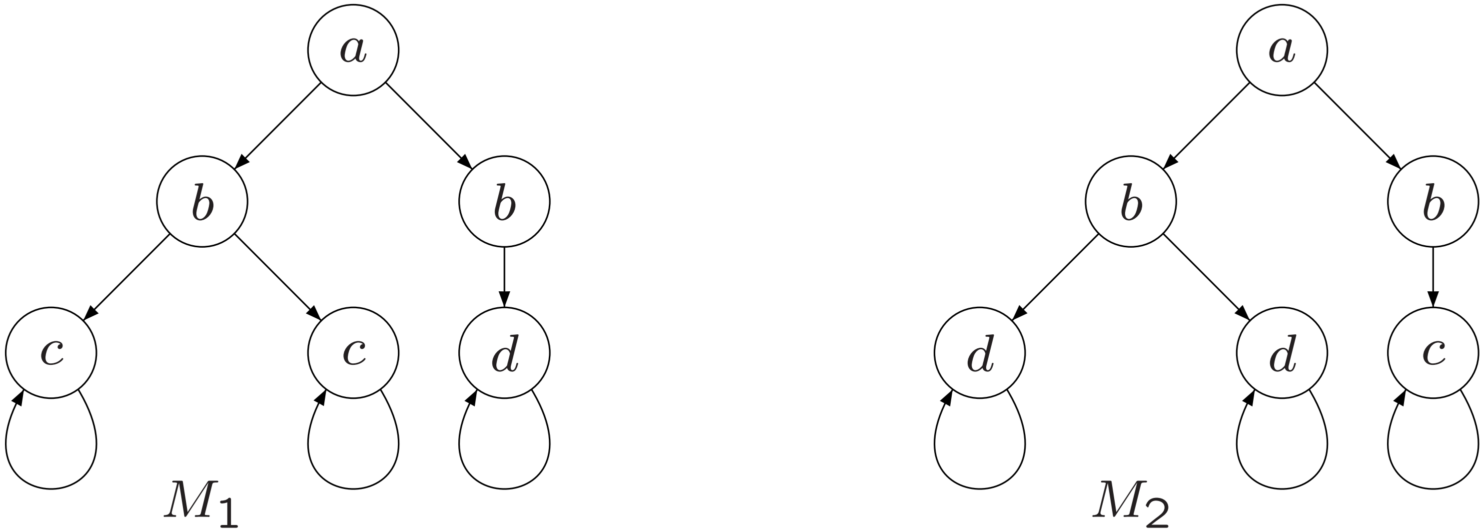
Generally: $M \simeq M' \Rightarrow M \preceq M' \wedge M' \preceq M$

In the figure: $M_1 \preceq M_2$, $M_2 \preceq M_1$ but $M_1 \not\simeq M_2$

Equivalent definition (as a game): $M \simeq M'$ if any choice of a model *and* of a move in it can be matched by an equally labelled move in the other model.

(choice of model done at each step $\Rightarrow$ symmetry)

Example: bisimulation



$M_1 \simeq M_2$
(duplicating nodes does not change branching properties)

Extension to fairness

The relation $\preceq_F \subseteq S \times S'$ is a *fair* simulation relation between M and M' (with $AP' \subseteq AP$) iff $\forall s \preceq_F s'$:
– $L(s) \cap AP' = L'(s')$
– for any *fair* path $\pi = s s_1 s_2 \ldots$ in M there exists a fair path $\pi' = s' s'_1 s'_2 \ldots$ in M' such that $\forall i > 0 \,.\, s_i \preceq s'_i$.
If $M \preceq_F M'$, then $\forall f \in ACTL*$, $M' \models_F f \Rightarrow M \models_F f$

The relation $\simeq_F \subseteq S \times S'$ is an *fair* bisimulation relation *echitabilă* between M and M' (with $AP' = AP$) iff $\forall s \simeq_F s'$:
– L(s) = L(s')
– for any *fair* path $\pi = s s_1 s_2 \ldots$ in M there exists a fair path $\pi' = s' s'_1 s'_2 \ldots$ in M' such that $\forall i > 0 \,.\, s_i \simeq s'_i$.
– for any *fair* $\pi' = s' s'_1 s'_2 \ldots$ in M' there exists a fair path $\pi = s s_1 s_2 \ldots$ in M such that $\forall i > 0 \,.\, s_i \simeq s'_i$.
If $M \simeq_F M'$, then $\forall f \in CTL*$, $M' \models_F f \Leftrightarrow M \models_F f$

Algorithms for checking bisimulation

Deterministic system = single initial state; any two successors differently labeled $s \rightarrow s_1 \wedge s \rightarrow s_2 \wedge s_1 \neq s_2 \Rightarrow L(s_1) \neq L(s_2)$

Simulation:

M, M' deterministic: $M \preceq M' \Leftrightarrow \mathcal{L}(M) \subseteq \mathcal{L}(M')$

In general, we recursively define: $s \preceq_0 s' \Leftrightarrow L(s) \cap AP' = L(s')$

$s \preceq_{n+1} s' \Leftrightarrow s \preceq_n s' \wedge \forall s_1 \,.\, s \rightarrow s_1 \Rightarrow \exists s_1' \,.\, s' \rightarrow s_1' \wedge s_1 \preceq_n s_1'$

We have $\preceq_{i+1} \subseteq \preceq_i \Rightarrow \exists n \,.\, \preceq_n = \preceq_{n+1} = \preceq$ (finite models)

Bisimulation:

M, M' deterministic: $M \simeq M' \Leftrightarrow \mathcal{L}(M) = \mathcal{L}(M')$

In general, we recursively define: $s \simeq_0 s' \Leftrightarrow L(s) = L(s')$

$$s \simeq_{n+1} s' \Leftrightarrow s \simeq_n s' \wedge \forall s_1 [s \rightarrow s_1 \Rightarrow \exists s_1' \,.\, s' \rightarrow s_1' \wedge s_1 \simeq_n s_1']$$
$$\wedge \forall s_1' [s' \rightarrow s_1' \Rightarrow \exists s_1 \,.\, s \rightarrow s_1 \wedge s_1 \simeq_n s_1']$$

We have $\simeq_{i+1} \subseteq \simeq_i \Rightarrow \exists n \,.\, \simeq_n = \simeq_{n+1} = \simeq$ (finite models)

Abstraction: Introduction

Abstraction is the key step in verifying systems of realistic size.

- it means constructing an *abstract* system (with fewer details)
- and establishing a *correspondence* between the abstract and the original system
  - exact abstractions: preserve truth value
  - conservative abstractions (approximations): correctness of abstract system implies correctness of real system, but not conversely (counterexample in the abstract system may not exist in the real one)

The abstract model must be obtained without building the concrete one
(the latter is often impossible due to size)

– *syntactic* abstraction techniques

– *semantic* abstraction techniques (e.g. reduced domain for variables)

Examples of encountered abstractions

Timed abstractions (region automaton; zone graph)
– are *finite* abstractions of an infinite-state systems
– several states in the concrete system match a state in the abstract system

A *specification* is usually an abstraction of the implementation
– the tableau for the LTL formula is an abstraction for a system that satisfies it

Refinement relations (language inclusion, simulation, etc.) between two different systems.

Using 1-bit packets in the protocol model of project 1 (data abstraction)

Cone of influence reduction

Abstraction by removal of variables that do not affect specification.

Let M be a system with variable set $V = \{v_1, v_2, \cdots, v_n\}$ described by the equations $v_i' = f_i(V)$.
Let V' be the set of variables referenced in the specification.
The *cone of influence* of V' = minimal set $C \subseteq V$ such that
– $V' \subseteq C$
– if $v_i \in C$, and f_i depends on v_j, then $v_j \in C$ (transitive closure)

We build a new system M' eliminating all the variables that do not appear in C, together with their functional equations.

Invariance of CTL* specifications

We prove that cone of influence reduction preserves the truth values of CTL* specifications (defined over variables from C).

Let $V = \{v_1, v_2, \cdots v_n\}$ be a set of *boolean* variables.
and $M = (S, S_0, R, L)$, with:
– $S = \{0, 1\}^n$ = set of assignments to V; $S_0 \subseteq S$
– $R = \bigwedge_{i=1}^{n}(v_i' = f_i(V))$
– $L(s) = \{v_i | s(v_i) = 1\}$ (variables equal to 1 in s)

Let V be numbered such that $C = \{v_1, \cdots, v_k\}$. We define $M' = (S', S_0', R', L')$:
– $S' = \{0, 1\}^k$ = set of assignments to C
– $S_0 = \{(d_1', \cdots, d_k') | \exists (d_1, \cdots d_n) \in S_0$ cu $d_1' = d_1 \wedge \ldots \wedge d_k' = d_k\}$
– $R' = \bigwedge_{i=1}^{k}(v_i' = f_i(C))$
– $L'(s) = \{v_i | s'(v_i) = 1\}$

We can show that the concrete model M and the abstract model M' are *bisimilar*.

Program slicing

A similar but more general notion for programs [Weiser'79]
– inspired by the mental processes performed during debugging
= calculating the program fragment that can affect the computed values in a given point of interest (slicing criterion) (e.g. variable at source line)
– usually: an *executable* program fragment, in source language
– based on program analysis notions of control and data dependence

Types of slicing:
– static or dinamic
– syntactic or semantic criteria
– forward or backward traversal of control graph
– type of control graph dependence: forward/backward; direct/transitive
– on *all* or *some* paths through control graph

Data abstraction

– used for reasoning about circuits with large bit width, or about programs with complex data structures
– useful if data processing operations are relatively simple (transfer, small number of arithmetic / logic ops)

Main idea: establishing a correspondence between original domain of data and a smaller-size domain (usually a few values)

Example: *sign abstraction*

$$h(x) = \begin{cases} - & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ + & \text{if } x > 0 \end{cases}$$

$\cdot$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

$+$	$-$	0	$+$
$-$	$-$	$-$	$\top$
0	$-$	0	$+$
$+$	$\top$	$+$	$+$

where $\top = \{-, 0, +\}$

$\Rightarrow$ we can not always have a precise abstraction
$\Rightarrow$ abstraction domain and function must be carefully chosen

Generating the abstract system

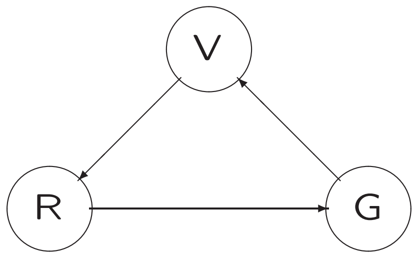
– for any variable x, we define an abstract variable $\widehat{x}$
– we label states with atomic propositions indicating the abstract value (for sign abstraction: 3 propositions p_x^-, p_x^0, p_x^+ for each variable x, indicating $\widehat{x} = "-"$, $\widehat{x} = 0$, $\widehat{x} = "+"$)
– we collapse all states with same abstract labels
$\Rightarrow$ abstract state space: 2^{AP}, AP = abstract propositions

For an *explicity* represented model M, we define the abstract (reduced) model $M_r = (S_r, S_r^0, R_r, L_r)$:
– $S_r = \{L_r(s) \mid s \in S\}$ = abstract labelings of states in S
– $S_r^0 = \{s_r^0 \in S_r \mid \exists s_0 \in S^0 \,.\, L_r(s_0) = s_r^0\}$ (labelings of initial states).
– $R_r(s_r, t_r) \Leftrightarrow \exists s, t \in S \,.\, R(s,t) \wedge L_r(s) = s_r \wedge L_r(t) = t_r$ (transitions between two abstract states if $\exists$ transitions between concrete representatives)

We can prove: abstract model M' simulates original (concrete) model M

Abstraction example

3-state traffic light reduced to 2 states



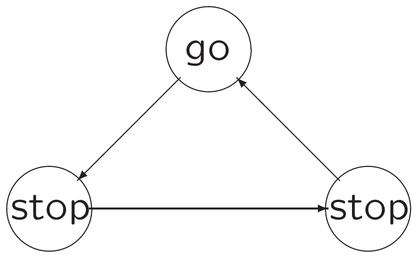
$L_r(R) = stop$

$L_r(G) = stop$

$L_r(V) = go$

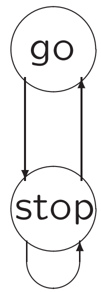
$\Rightarrow$

relabeling



$\Rightarrow$

collapsing



Note: the abstract system may introduce new behaviors (e.g., the system can stay in the "stop" state forever).

Exact and approximate abstractions

Consider a system represented *implicitly*, by predicates for the transition relation $\mathcal{R}$ and the initial states $\mathcal{S}_0$.
We assume the same abstraction function for all variables,
$h : D \to A$ (D = concrete domain, A = abstract domain)

We must define $\widehat{\mathcal{S}_0}$ and $\widehat{\mathcal{R}}$ for the abstract system:
$\widehat{\mathcal{S}_0} = \exists x_1 \ldots \exists x_n \,.\, \mathcal{S}_0(x_1, \cdots, x_n) \wedge h(x_1) = \widehat{x_1} \wedge \cdots \wedge h(x_n) = \widehat{x_n}$
We similarly define $\widehat{\mathcal{R}}(\widehat{x_1}, \cdots \widehat{x_n}, \widehat{x_1}', \cdots \widehat{x_n}')$.
$\Rightarrow$ from $\phi(x_1, \cdots, x_n)$ we obtain $\widehat{\phi}(\widehat{x_1}, \cdots, \widehat{x_n})$ expressed in abstract variables

Transforming $\phi \to \widehat{\phi}$ may be a complex operation $\Rightarrow$ we apply it (like negation) just to elementary relations between variables (e.g., $=, <, >$, etc.).
Define by structural induction an approximate abstraction $\mathcal{A}$:
– $\mathcal{A}(P(x_1, \ldots, x_n)) = \widehat{P}(\widehat{x_1}, \cdots, \widehat{x_n})$, if P is an elementary relation.
– $\mathcal{A}(\neg P(x_1, \ldots, x_n)) = \neg \widehat{P}(\widehat{x_1}, \cdots, \widehat{x_n})$
– $\mathcal{A}(\phi_1 \wedge \phi_2) = \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$ – $\mathcal{A}(\phi_1 \vee \phi_2) = \mathcal{A}(\phi_1) \vee \mathcal{A}(\phi_2)$
– $\mathcal{A}(\exists x\, \phi) = \exists \widehat{x}\, \mathcal{A}(\phi)$ – $\mathcal{A}(\forall x\, \phi) = \forall \widehat{x}\, \mathcal{A}(\phi)$

Exact and approximate abstractions (cont'd)

With the definitions so far, one can prove: $\forall \phi \,.\, \widehat{\phi} \Rightarrow \mathcal{A}(\phi)$
In particula, $\widehat{\mathcal{S}_0} \Rightarrow \mathcal{A}(\mathcal{S}_0)$ and $\widehat{\mathcal{R}} \Rightarrow \mathcal{A}(\mathcal{R})$.
(approximation may introduce additional initial states and transitions)

Fie modelul abstract aproximat $M_a = (S_r, \mathcal{A}(\mathcal{S}_0), \mathcal{A}(\mathcal{R}), L_r)$. Then $M \preceq M_a$ (the abstract approximated model simulates the original)

If the abstraction function preserves the relations which corresponds to primitive operations in a program, the abstraction $\mathcal{A}$ is exact.

An abstraction function h_x defines an equivalence relation between the concrete values for x which correspond to the same abstract values:

$$d_1 \sim_x d_2 \Leftrightarrow h_x(d_1) = h_x(d_2)$$

If the value of any primitive relation P in the program is the same for any two pair of equivalent concrete values:

$$\forall d_1, \cdots d_n, d'_1, \cdots d'_n \,.\, \bigwedge_{i=1}^{n} d_i \sim_{x_i} d'_i \Rightarrow P(d_1, \cdots, d_n) = P(d'_1, \cdots, d'_n)$$

then $M \simeq M_a$ (the abstract model simulates the concrete model)

Abstract interpretation

A method for defining the *abstract semantics* of a program that can be used to analyse the program and produce information about its runtime behavior. [Cousot & Cousot '77]

Consists in:
– a concrete domain D and an abstract domain A, linked via a *Galois connection*:
– an abstraction function $\alpha : D \to A$
– a concretization function $\gamma : A \to \mathcal{P}(D)$
(associates to each abstract state a set of concrete states)
– a.î. $\forall x \in \mathcal{P}(D) \,.\, x \subseteq \gamma(\alpha(x))$ și $\forall a \in A \,.\, a = \alpha(\gamma(a))$
(abstraction followed by concretization introduces approximation)
concretization followed by abstraction is exact

the majority of abstractions can be formulated in this general framework

Example: Abstractions modulo an integer

For arithmetic circuits/programs, the abstraction defined by:

$$h(x) = x \bmod n,\ n \in \mathbf{Z}$$

Preserves primitive mathematical relations, because

$$((x \bmod n) + (y \bmod n)) \bmod n = (x + y) \bmod n, \text{ etc.}$$

Additionally (chinese remainder theorem): if $n_1, \cdots n_k$ relatively prime, and $n = n_1 \cdot n_2 \cdot \ldots \cdot n_k$, then

$$x \equiv y \pmod{n} \Leftrightarrow \bigwedge_{i=1}^{k} x \equiv y \pmod{n_i}$$

$\Rightarrow$ to verify 16-bit arithmetic, it suffices to verify the implementation for integers modulo 5, 7, 9, 11, 32 (product $>$ 2^{16})

Symbolic abstractions

To verify the datapaths of a system
(main function: computing and preserving values)

Example: correct transmission from a to b. Initially, for a fixed value:

$$\mathbf{AG}(a = 17 \rightarrow \mathbf{AX}\, b = 17)$$

Abstraction function: $h(x) = \begin{cases} 1 & \text{if } x = 17 \\ 0 & \text{otherwise} \end{cases}$

More generally: we introduce the symbolic parameter c:

$$h(x) = \begin{cases} 1 & \text{if } x = c \\ 0 & \text{otherwise} \end{cases}$$

$\Rightarrow$ abstract transition relation $\widehat{R}(\widehat{a}, \widehat{a}', \widehat{b}, \widehat{b}', c)$

In a BDD representation, c does not affect the complexity if the system behavior does not depend on c

Example: pipelined adder with two stages

$$\mathbf{AG}(reg1 = a \wedge reg2 = b \rightarrow \mathbf{AX}\,\mathbf{AX}\, sum = a + b)$$

Compozitional reasoning

an application of "divide and conquer" to verification of a system built from components
– verification of local properties of components
– deriving global properties from component properties
– without constructing a model of the entire system (impractical)

Compositional reasoning: generic term for rules of the form
– $M_1 \models f_1 \wedge M_2 \models f_2 \Rightarrow Compose(M_1, M_2) \models LogicOp(f_1, f_2)$
e.g. parallel composition, and $LogicOp = \wedge$
– $M_1 \prec M_2 \Rightarrow CompOp(M_1) \prec CompOp(M_2)$
ex. $\prec$ = implementation, refinement; $CompOp(\cdot) = \cdot || M$
– $M_1 \prec S_1 \wedge M_2 \prec S_2 \Rightarrow Compose(M_1, M_2) \prec Compose(S_1, S_2)$

CSynchronous composition, simulation and fair ATCL

Let $M = (S, S_0, AP, L, R, F)$ and $M' = (S', S'_0, AP', L', R', F')$.
Define parallel synchronous composition $M'' = M||M'$:

- $S'' = \{(s, s') \in S \times S' \mid L(s) \cap AP' = L'(s') \cap AP\}$
- $S''_0 = (S_0 \times S'_0) \cap S''$
- $AP'' = AP \cup AP'$
- $L''(s, s') = L(s) \cup L'(s')$
- $R''((s, s')(t, t')) = R(s, t) \wedge R'(s', t')$
- $F'' = \{(P \times S') \cap S'' \mid P \in F\} \cup \{(S \times P') \cap S'' \mid P' \in F'\}$

We use ACTL with fairness: for any ACTL formula f we can construct a tableau $\mathcal{T}_f$, and we have $M \models_F f \Leftrightarrow M \preceq_F \mathcal{T}_f$
$\Rightarrow$ we can reason uniformly with formulas and models (tableaux)

(a) for any M și M', $M||M' \preceq_F M$.
(b) for any M, M' și M'', $M \preceq_F M' \Rightarrow M||M'' \preceq_F M'||M''$
(c) for any M, $M \preceq_F M||M$

Non-circular assume-guarantee

Folosim notația $\langle f \rangle M \langle g \rangle$:
Orice sistem care satisface prezumția f și conține M garantează g.
(f, g sunt fie formule, fie modele)

O structură tipică de raționament:
$\langle \textit{true} \rangle M \langle A \rangle \wedge \langle A \rangle M' \langle g \rangle \wedge \langle g \rangle M \langle f \rangle \Rightarrow \langle \textit{true} \rangle M || M' \langle f \rangle$
Instanțiere în termeni concreți:
M = un transmițător complex
A = un model simplu de transmițător periodic
  $\langle \textit{true} \rangle M \langle A \rangle$: M funcționează la fel ca și A
M' = un receptor
g = "mesajele sunt preluate la timp"
  $\langle A \rangle M' \langle g \rangle$ = M' compus cu A preia mesajele la timp
f = "nu avem buffer overflow"
  $\langle g \rangle M \langle f \rangle$ = dacă M e într-un sistem care preia mesajele la timp,
  nu avem buffer overflow.
$\Rightarrow$ în sistemul $M || M'$ nu apare buffer overflow.

Justificarea raționamentului

(1) $M \preceq_F A$	ipoteză
(2) $M\|\|M' \preceq_F A\|\|M'$	(1) și compoziționalitate (a)
(3) $A\|\|M' \models_F g$	ipoteză
(4) $A\|\|M' \preceq_F \mathcal{T}_g$	(3) și prop. tabloului ACTL
(5) $M\|\|M' \preceq_F \mathcal{T}_g$	(2), (4) și tranzitivitatea $\preceq_F$
(6) $M\|\|M\|\|M' \preceq_F \mathcal{T}_g\|\|M$	(5) și compoziționalitate (b)
(7) $\mathcal{T}_g\|\|M \models_F f$	ipoteză
(8) $M\|\|M\|\|M' \models_F f$	(6), (7) și $\preceq_F \Rightarrow \models_F$
(9) $M \preceq_F M\|\|M$	compoziționalitate (c)
(10) $M\|\|M' \preceq_F M\|\|M\|\|M'$	(9) și compoziționalitate (b)
(11) $M\|\|M' \models_F f$	(8), (10) și $\preceq_F \Rightarrow \models_F$

Demonstratoare de teoreme pot mecaniza descompunerea
în raționamente pe componente și asigura validitatea deducției.

Circular assume-guarantee

Often, compositional rules are not strong enough.
Consider implementations M_i and specifications S_i, $i = 1, 2$.
To prove $M_1 || M_2 \prec S_1 || S_2$ it would suffice if $M_1 \prec S_1$ and $M_2 \prec S_2$.
But frequently, these individual relations are not satisfied:
– components M_1 and M_2 are not independently designed
– each one relies on functioning in an environment provided by the other one

Exemplu de dependențe

Modelăm algoritmul obișnuit de împărțire a două numere, $n \div d$, în baza b, cu două componente:

$M_Q(in : r, d; out : q)$ calculează următoarea cifră din cât: $q = \lfloor r/d \rfloor$

$M_R(in : n, d, q; out : r)$ actualizează restul: $r' = (r - q * d) * b + next\_digit(n)$

Dorim ca $M_Q || M_R$ să satisfacă împreună următorii invarianți:

- S_Q: $0 \leq q < b \wedge q * d \leq r < (q + 1) * d$
- S_R: $0 \leq r < b * d$

Totuși, individual nu avem nici $M_Q \models S_Q$ și nici $M_R \models S_R$: funcționarea corectă a fiecărui modul depinde de celălalt

Dar avem $S_Q \Rightarrow M_R \models S_R$ și $S_R \Rightarrow M_Q \models S_Q$.

(un modul funcționează corect în mediul dat de *specificarea* celuilalt)

$\Rightarrow$ Putem deduce de aici că $M_Q || M_R \models S_Q \wedge S_R$?

Circular assume-guarantee rules

Studied in various contexts [Chandi & Misra'81, Abadi & Lamport'93]

We refer to Reactive Modules [Alur & Henzinger '95]:
– modules with input and putput variables, and transition relation
– dependence relation $\prec \subseteq (V_{in} \cup V_{out}) \times V_{out}$
– $x \prec y$: y depends *combinaționally* on x;
otherwise, only the next value of y can depend sequentially on x
– synchronous parallel composition $M_1 || M_2$ is possible
if $V_{out}(M_1) \cap V_{out}(M_2) = \emptyset$ and $\prec_{M_1} \cup \prec_{M_2}$ is an acyclic relation

We define the *refinement* (implementation) relation $M \leq M'$ iff
$V(M') \subseteq V(M)$, $V_{out}(M') \subseteq V_{out}(M)$, $\prec_M \supseteq \prec'_M$, $\mathcal{L}(M)|_{V(M')} \subseteq \mathcal{L}(M')$
(first 3 conditions: if P can function in a context, so can Q)

Circular assume-guarantee rules (cont'd)

For reactive modules:
$$\frac{\begin{array}{l} M_1 || S_2 \leq S_1 || S_2 \\ S_1 || M_2 \leq S_1 || S_2 \end{array}}{M_1 || M_2 \leq S_1 || S_2}$$
(assuming all compositions well defined)

Advantage: although there are two relations to prove, each is simpler than the original one.
– specification description S_i is much simpler than the implementation M_i
– need not compose two different implementations (often impossible)

Rule with temporal induction [McMillan'97]
valid for *invariants* (safety properties)
– if $P_1 \wedge Q_1$ true at $0, 1, \cdots, t \Rightarrow Q_2$ true at $t + 1$
– if $P_2 \wedge Q_2$ true at $0, 1, \cdots, t \Rightarrow Q_1$ true at $t + orice$
– then for any t, $P_1 \wedge P_2 \Rightarrow Q_1 \wedge Q_2$

Compositionality and refinement

[Henzinger'01] - study of the theory of interfaces

For a refinement relation $\leq$ and a composition relation $||$, we wish:

If $M_1 \leq S_1$ and $M_2 \leq S_2$, then $M_1||M_2 \leq S_1||S_2$

Generally, insufficient – components may be incompatible.

$\Rightarrow$ two variants:

• If $M_1 \leq S_1$ and $M_2 \leq S_2$, and $M_1||M_2$ is defined,
then $S_1||S_2$ is defined and $M_1||M_2 \leq S_1||S_2$
– formalism focused on *components*
– allows independent verification of components (bottom-up)

• If $M_1 \leq S_1$ and $M_2 \leq S_2$, and $S_1||S_2$ is defined,
then $M_1||M_2$ is defined and $M_1||M_2 \leq S_1||S_2$
– formalism focused on *interfaces*
– allows independent implementation of interfaces (top-down)