Assume-Guarantee Compositional Reasoning

Marius Minea

"Politehnica" University of Timișoara, Romania

CS^2, Szeged, June 29, 2006

Talk outline

- *Compositional reasoning and circular assume guarantee*
- Assume-guarantee for hierarchical hybrid systems
- Compositional safety interfaces
- Compositionality in timed systems: survey and research agenda

Compositional reasoning: Motivation

Systems are complex $\Rightarrow$ need to apply "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 $LogicOp = \wedge$
$M_1 \models f_1 \wedge M_2 \models f_2 \Rightarrow \quad M_1 || M_2 \models f_1 \wedge f_2$
– $M_1 \prec M_2 \Rightarrow CompOp(M_1) \prec CompOp(M_2)$
ex. $\prec$ = implementation, refinement; $\quad CompOp(\cdot) = \cdot || M$
$M_1 \prec M_2 \Rightarrow \quad M_1 || M \prec M_2 || M$
– $M_1 \prec S_1 \wedge M_2 \prec S_2 \Rightarrow Compose(M_1, M_2) \prec Compose(S_1, S_2)$

The limitations of compositionality

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 relies on functioning in an environment provided by the other

Example:

specifications: $S_1 : x = 0;$ $S_2 : y = 0$ (invariant)

modules: $M_1 : x_0 = 0; next(x) = y;$ $M_2 : y_0 = 0; next(y) = x;$

We have $M_1 || M_2 \prec S_1 || S_2$ but $M_1 \not\prec S_1$, $M_2 \not\prec S_2$

But in the right context: $M_1 || S_2 \prec S_1$ and $M_2 || S_1 \prec S_2$

Non-circular assume-guarantee

Familiar case: Hoare rules/triples for sequential programs:

$$\{P\} \quad S \quad \{Q\}$$

P : $precondition$; S: statement; Q: postcondition

In practice, one can use pre/postconditions at procedure boundaries
– *intraprocedural* analysis to establish/check individual pre/postconditions
– *interprocedural* analysis starting with given pre/postconditions for a full program check

– languages with built-in assume-guarantee support
(Eiffel: "design by contract")
– add-ons, e.g. JML for Java (used by ESC/Java static analyzer)

```
/*@ non_null */ int[] a;
//@ invariant 0 <= n && n <= a.length;
//@ requires input != null;            ... etc.
```

Circular assume-guarantee rules

Ideally, we'd like a rule of the form:

$$\frac{\begin{array}{ccc}\{P_2\} & M_1 & \{P_1\} \\ \{P_1\} & M_2 & \{P_2\}\end{array}}{\{true\} \;\; M_1 || M_2 \;\; \{P_1 \wedge P_2\}}$$

(M_1 guarantees P_1 provided that M_2 guarantees P_2 and vice versa)
– is NOT generally sound !

Circular AGR originates with [Chandi & Misra'81, Jones '83]
[Abadi & Lamport '93, '95] (Composing/Conjoining Specifications)

Circular assume-guarantee rules

We refer to Reactive Modules [Alur & Henzinger '95]:
– modules with input and output variables, and transition relation
– dependence relation $\prec \subseteq (V_{in} \cup V_{out}) \times V_{out}$
– $x \prec y$: y depends *combinationally* 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 usually simpler than implementation M_i

– need not compose two different implementations (often impossible)

Rule with temporal induction [McMillan'99]

Induction over (discrete) time steps is crucial to proving soundness of assume-guarantee rules

– e.g., for reactive modules, proof uses double induction:
  over sequence of sub-steps (variables that change combinationally)
  over sequence of steps (length of execution trace)

McMillan ('99) states an explicit temporal induction rule 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 $t + 1$
– 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)

Practical issues

– Tool support

e.g. Mocha [Berkeley/UPenn]: support for proof decomposition using assume-guarantee proofs; also proof manager

LTSA: assumptions modeled as finite-state automata

– Completeness of assume-guarantee rules

given a system composed of (two) models, are there always environments that can be used in a circular AGR rule ? How can they be found ? [Namjoshi & Trefler '00];
L* learning approach [Giannakopoulou, Pasareanu et al.]

– Automated decomposition

How to choose decomposition boundaries in a complex system ?

Talk outline

- Compositional reasoning and circular assume guarantee
- *Assume-guarantee for hierarchical hybrid systems*
- Compositional safety interfaces
- Compositionality in timed systems: survey and research agenda

Assume-guarantee reasoning for hierarchical hybrid systems

[T. A. Henzinger, M. Minea, V. Prabhu, HSCC 2001]

Goal: synthesis of hybrid systems by top-down refinement
with verification supported by design flow

Achieved through:

- A formal model for hierarchical hybrid systems
- with compositional semantics
- and refinement checking by assume-guarantee reasoning

Masaccio: formal hybrid components [Henzinger '00]

A formal model inspired from:

- Reactive Modules (discrete behavior and composition)
- Hybrid Automata (continuous and real-time behavior)

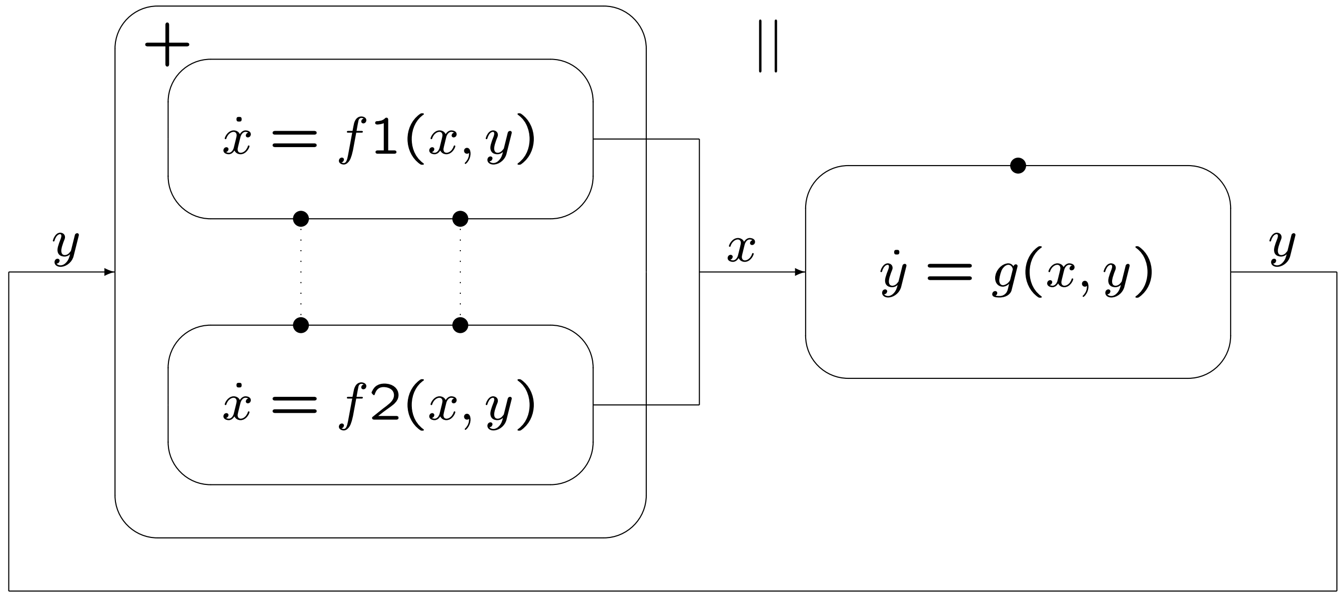
Enhancements:

- Parallel and serial composition, arbitrarily nested
- Discrete and continuous dynamics, arbitrarily composed

Sample Masaccio Model

Example: plant g and controller with modes $f1$ and $f2$

– components with parallel and serial composition (Statecharts-like)
– explicit flow of control + math. equations for continuous quantities



Components in Masaccio

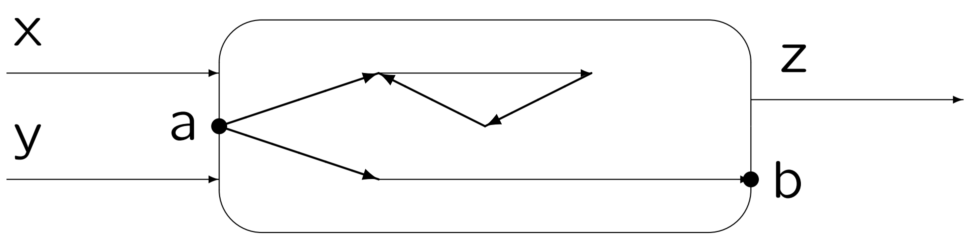
- Component = interface + behavior
- Interface: interaction with other components

– Data: variables (input/output, discrete/continuous)

  – dependence relation: $x \prec y$

  for combinational await dependency $y' = f(x')$

– Control: locations, with entry conditions on data variables



- Behavior: set of executions

– Jumps: instantaneous change of variables $(\bar{x}, \bar{x}')$,

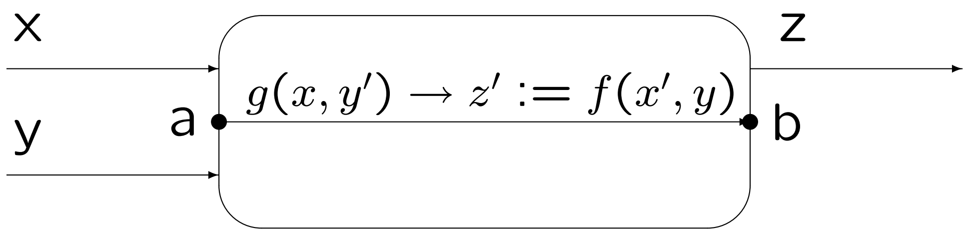
– Flows: evolution of continuous variables:

  (f, δ) with function f and real-valued duration δ

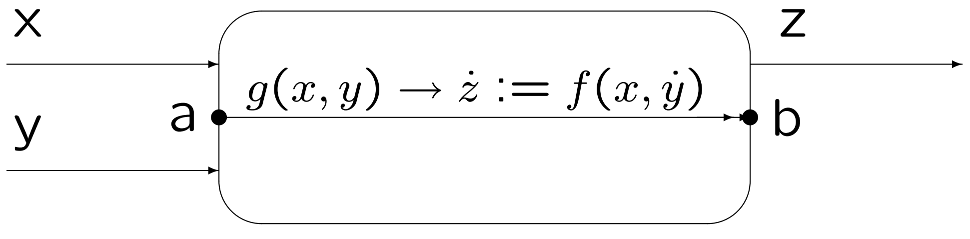
Execution: $(a, s_1 s_2 \cdots s_n, b)$ or $(a, s_1 s_2 \cdots)$, with s_i jumps or flows

Atomic Components

Atomic discrete component: guarded difference equation



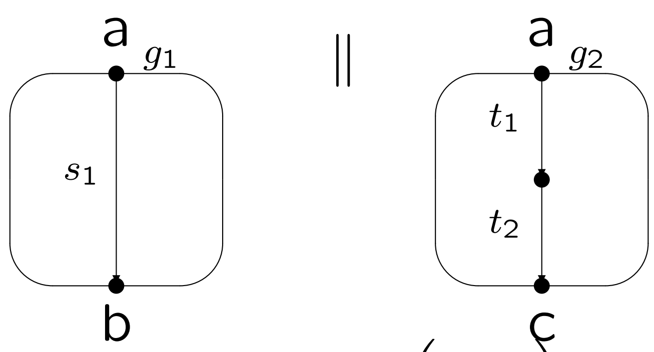
Atomic continuous component: guarded differential equation



+ Component operations: composition, renaming, hiding

Operations: Parallel Composition

Execution: (a, s_1, b)



Execution: $(a, t_1 t_2, c)$

Execution: $(a, \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, b)$

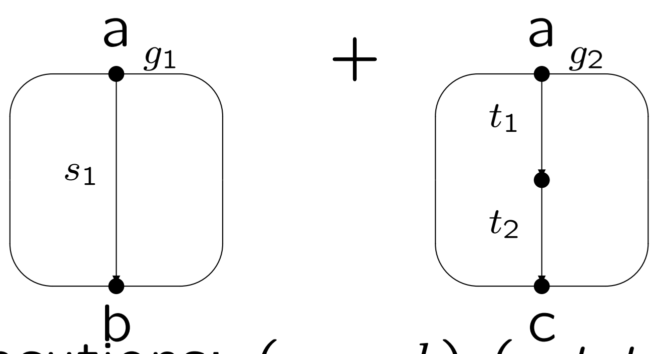
- synchronous conjunction of component behaviors

jumps correspond to jumps, and flows to flows of same duration

- same entry locations and projections of entry conditions
- union of dependence relations: acyclic
- one component may preempt another

Operations: Serial Composition

Execution:
(a, s_1, b)



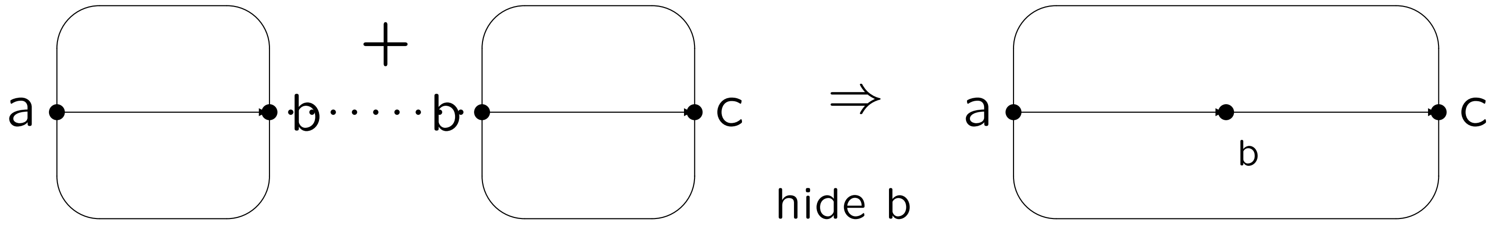
Execution:
$(a, t_1 t_2, c)$

Executions: $(a, s_1, b), (a, t_1 t_2, c)$

- disjunction of component behaviors
- entry condition determines component that executes
- can represent different execution modes

Operations: Hiding and Renaming

- Location hiding: makes location internal to a component
- strings together component executions
- hidden location has entry condition *true* $\Rightarrow$ avoids deadlock
- no-op jumps always possible at hidden locations
- used with serial composition



- Variable hiding
- Location and variable renaming

Refinement in Masaccio

Trace inclusion: not satisfactory

Generally: A $<$ B means "A is more specific than B"
Parallel composition:
if A $=$ B $\parallel$ C then A $<$ B (B is projection of A)

Serial composition:
if A $=$ B $+$ C then A $<$ B (B is prefix of A)

Formally: A $<$ B if every trace (a, w, c) or (a, w) of A
– is either a trace of B
– or has a prefix (a, w', b) which is a trace of B

Compositionality

All component operations are compositional w.r.t. refinement:

- $A < B \Rightarrow A + C < B + C$ serial composition
- $A < B \Rightarrow A \parallel C < B \parallel C$ parallel composition
- $A < B \Rightarrow A \setminus a < B \setminus a$ location hiding
- $A < B \Rightarrow A\ [a{:=}b] < B\ [a{:=}b]$ location renaming
- $A < B \Rightarrow A \setminus x < B \setminus x$ data hiding
- $A < B \Rightarrow A\ [x{:=}y] < B\ [x{:=}y]$ data renaming

More generally, for any context C:

$$A < B \Rightarrow C[A] < C[B]$$

context = component expression with placeholder

e.g. $C[\cdot] = \cdot \parallel D + E$

Circular Assume-Guarantee Reasoning

$$\frac{\begin{array}{c} A1 \parallel B2 < A2 \parallel B2 \\ A2 \parallel B1 < A2 \parallel B2 \end{array}}{A1 \parallel B1 < A2 \parallel B2}$$

$A1 < A2$ only in the *context* of B2, etc.

– requires several conditions for circularity to be sound

– typically applicable only to *safety* properties

– *nonblocking* conditions: environment B2 may not block A1

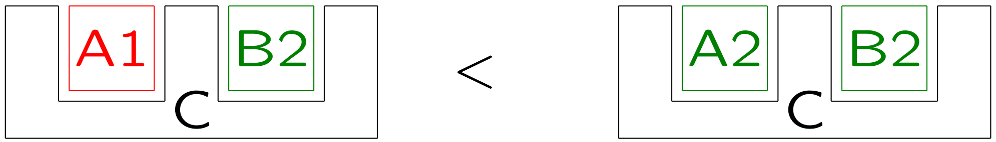
– typically used for *parallel* composition; for serial case: [Alur & Grosu '00]

Assume-Guarantee in Masaccio

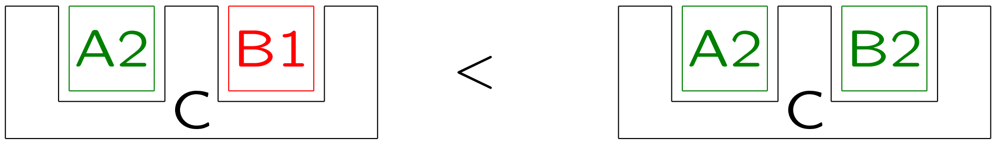
Refinement goal: context with two implementation components

Premises: individually replace components with specification

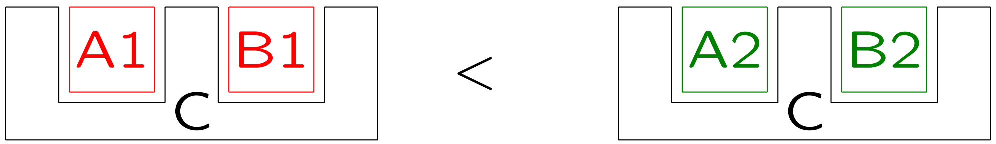
$C[A1,B2] < C[A2,B2]$



$C[A2,B1] < C[A2,B2]$

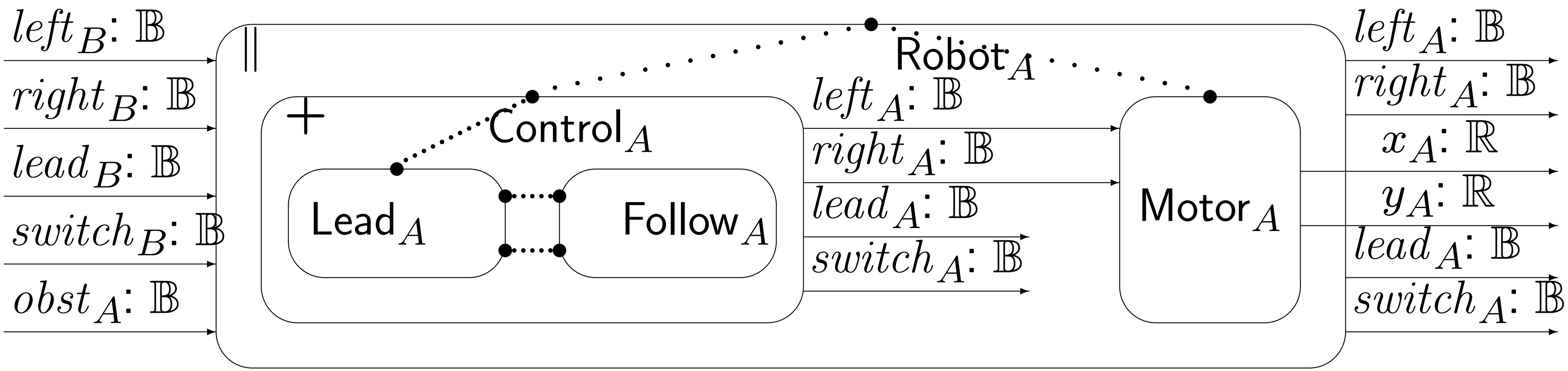


$C[A1,B1] < C[A2,B2]$



Example: Communicating Robots

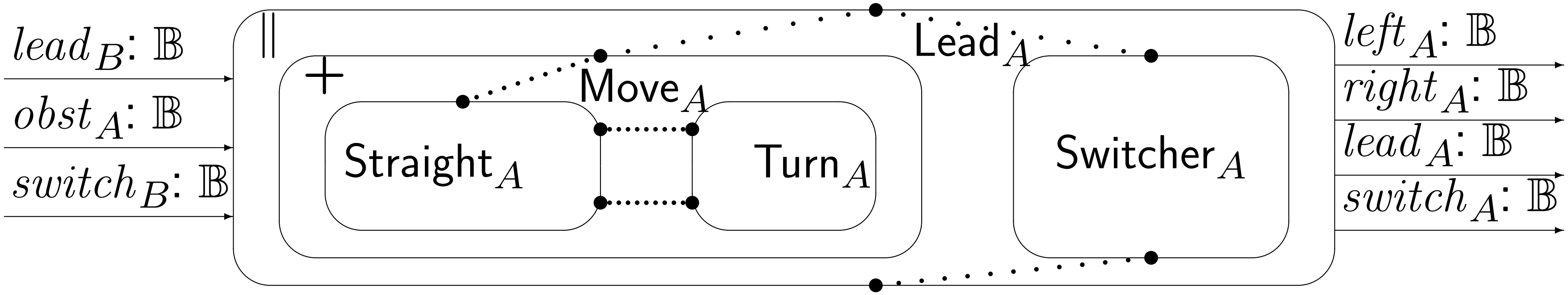
Two robots alternate leading and following



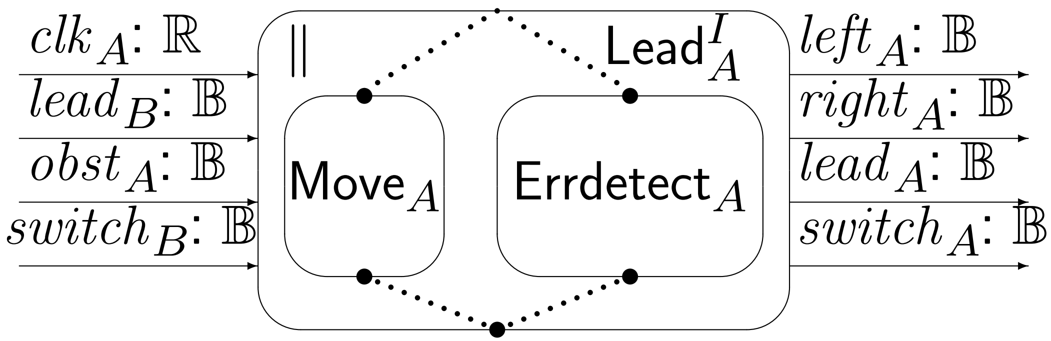
- robot in follow mode mimics robot in lead mode
- mode switch upon hitting obstacle or at random

Refinement of Robot Synchronization

Leading robot goes straight or turns around obstacle:



Implement more robust switching from lead to follow:
Error detection component takes place of switcher



Applying Assume-Guarantee

Need to prove:

$$C_A[\mathsf{Control}^I_A] || C_B[\mathsf{Control}^I_B] < C_A[\mathsf{Control}_A] || C_B[\mathsf{Control}_B]$$

With assume-guarantee:

$$C_A[\mathsf{Control}^I_A] || C_B[\mathsf{Control}'_B] < C_A[\mathsf{Control}_A] || C_B[\mathsf{Control}'_B]$$
$$C_A[\mathsf{Control}_A] || C_B[\mathsf{Control}^I_B] < C_A[\mathsf{Control}_A] || C_B[\mathsf{Control}_B]$$

By compositionality:

$$\mathsf{Control}^I_A \parallel \mathsf{Control}'_B < \mathsf{Control}_A \parallel \mathsf{Control}'_B$$
$$\mathsf{Control}_A \parallel \mathsf{Control}^I_B < \mathsf{Control}_A \parallel \mathsf{Control}_B$$

Talk outline

- Compositional reasoning and circular assume guarantee
- Assume-guarantee for hierarchical hybrid systems
- *Compositional safety interfaces*
- Compositionality in timed systems: survey and research agenda

Compositional Safety Interfaces

[jointly with Jonas Elmqvist and Simin Nadjm-Tehrani, U. Linköping]

Context: component-based development of safety-critical systems

Question: how to characterize a component ?
– behavior in the "intended" environment
– behavior in the presence of single / multiple faults

Two roles:
– component developer establishes safety interface
– component integrator performs safety analysis
(requiring only safety interfaces, not full component descriptions)

Fault Models

Component model: reactive modules [Alur & Henzinger], with input / output / private variables V_i, V_o, V_p.

To model input faults $\Rightarrow$ input v_i of model M no longer controlled by environment of M, but by a *fault module*.

Fault module F for M: one input v_i^f, one output v_i, unconstrained transition relation (but could be specialized).

We might regard the fault as:
– composed with the module M
– composed with the environment E of M: $F_i \circ E = F_i || E[v_j / v_j^f]$

Satisfying Environment

Our problem:
given module M and system safety property φ,
in what environment (of other components) must M be placed
for the global system to satisfy φ ?

(assuming no faults, or in the presence of specific single/double faults)

Observation: if $M \models \varphi$, then $M||E \models \varphi$

Else, if $M \not\models \varphi \Rightarrow$ iterative generation of satisfying environment E:
– model check $M||E_i \models \varphi$ and find counterexample
– restrict E_i to E_{i+1} to eliminate counterexample
– iterate to fixpoint

Done experimentally using tools for synchronous languages (Esterel and SCADE/Lustre)

Safety Interfaces

Given a module M, a system-level safety property φ, a safety interface S^φ for M is a tuple $\langle E^\varphi, \mathsf{single}, \mathsf{double}\rangle$ where

- E^φ is an environment in which $M \parallel E^\varphi \models \varphi$.
- $\mathsf{single} = \langle F^s, E^s\rangle$ where $F^s \subseteq \mathcal{P}(F)$ is a set of faults (the *single fault resilience set* and) E^s is an environment such that $\forall F_k \in F^s$ $M \parallel (F_k \circ E^s) \models \varphi$
- $\mathsf{double} = \{\langle F^d_1, E^d_1\rangle, \ldots, \langle F^d_n, E^d_n\rangle\}$ with $F^d_k = \langle F^1_k, F^2_k\rangle$, F^1_k,$F^2_k \in F$, $F^1_k \neq F^2_k$ such that $M \parallel ((F^1_k \parallel F^2_k) \circ E^d_k) \models \varphi$

$\Rightarrow$ safety interface characterizes satisfying environments for M and ϕ in the presence of up to double faults

Goal: reason about composed system using *only* safety interfaces

An n–module Assume-Guarantee Rule

Let M_j and E_j be modules and environments such that the compositions $I = M_1 \parallel \ \ldots \parallel \ M_n$ and $E = E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$ exist and $V_j^E \subseteq V_{obs}^I$.
Then, if $\forall j \forall k \ M_j \parallel \ E_j \leq E_k$ we have $M_1 \parallel \ \ldots \parallel \ M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n$.
(where $\widehat{\parallel}$ denotes nonblocking parallel composition)

more succinctly: $\dfrac{\forall j \forall k \ M_j \parallel \ E_j \leq E_k}{M_1 \parallel \ \ldots \parallel \ M_n \leq E_1 \mathbin{\widehat{\parallel}} \ldots \mathbin{\widehat{\parallel}} E_n}$

In other words: each module M_j when placed in its needed environment E_j refines the needed environment for each other module M_k

For specifications φ: $\dfrac{\forall j \ M_j \parallel \ E_j \models \varphi \qquad \forall j \forall k \ M_j \parallel \ E_j \leq E_k}{M_1 \parallel \ M_2 \parallel \ \ldots \parallel \ M_n \models \varphi}$

Assume-Guarantee for Faults

Single faults:
– a module in any environment (even faulty one) still provides an environment that guarantees the safety of each other module in absence of another fault

$$M_i \parallel E_i^{\varphi} \leq E_k^{\varphi}$$

– a module in a non-faulty environment provides for every other module an environment which makes it resilient to single faults.

$$M_k \parallel E_k^{\varphi} \leq E_i^{s}$$

Double faults: similar (three types of rules);
some premises common or subsume those for single faults

Experimental results: model of aircraft leakeage detection system
– compositional analysis for single and double faults $\Rightarrow$ system safe

Talk outline

- Compositional reasoning and circular assume guarantee
- Assume-guarantee for hierarchical hybrid systems
- Compositional safety interfaces
- *Compositionality in timed systems: survey and research agenda*

Modularity for Timed and Hybrid Systems

[Alur, Henzinger 1997]
– modularity, liveness and control in reactive and real-time setting
– discuss the case of *open* systems
– extend formalism of *reactive modules* to real-time

– receptiveness condition becomes *nonzenoness* (diverging time)
– analyze it as game between system and environment (both symbolic and region-graph algorithm), extending timed I/O automata results
– circular assume-guarantee rule remains valid for receptive modules:

$$P_1||Q_2 \leq Q_1 \wedge Q_1||P_2 \leq Q_2 \Rightarrow P_1||P_2 \leq Q_1||Q_2$$

– use results for synthesis of receptive controllers

Simulation and Assume-Guarantee for TA

[Serdar Tasiran, PhD thesis, Berkeley, 1998]

1) Checking *timed refinement* (timed trace inclusion/timed simulation)
– gives algorithm using homomorphisms and reduction to checking of untimed homomorphism
– relies on region graph construction, can quickly become complex

2) Assume-guarantee reasoning for timed abstractions ($\leq_L$ and $\leq_S$)
– requires *non-blocking* timed automata: react to any input, and outputs change due to inputs only after non-zero delay
– with these restrictions, circular assume-guarantee applies:

if $A_1 || B_2 \leq_L A_2$ and $A_2 || B_1 \leq_L B_2$ then $A_1 || B_1 \leq_L A_2 || B_2$

– same rule with same conditions applies for timed simulation $\leq_S$
– witness simulation for composition: computed from simulation relations for components

Assume-Guarantee for Timing Diagrams

[Amla, Emerson, Namjoshi, Trefler 2001] "timing" in diagrams is not explicit, but implicit in a reference clock
– generic formalism for synchronous composition of processes with variables
– to deal with liveness: need *closure* $CL(P)$ of process P
– prior approach [Alur & Henzinger '96] breaks circularity by taking closure of specification in one assumption: $CL(Q_1)||P_2 \models Q_2$
– here: additional check; can still use liveness properties as assumptions

Assumptions for $P_1||P_2 \models S$:
– $P_1||Q_2 \models Q_1$ and $Q_1||P_2 \models Q_2$ and $Q_1||Q_2 \models$S($spec$)
– $P_1||CL(T) \models T + Q_1 + Q_2$ or $P_2||CL(T) \models T + Q_1 + Q_2$

Timing diagrams are formalizations of those used in circuit descriptions (with clock waveforms, sequential and concurrent dependencies)
– could timing constraints be added ?

Timed Interfaces

[de Alfaro, Henzinger, Stoelinga 2002]
– specify both *assumptions* (about timing of inputs) as well as *guarantees* (about timing of outputs)
– semantics is *optimistic*: an interface is *well-formed* if there is at least *some* environment that satisfies its input assumptions
– similarly, interfaces are *compatible* iff composition is *well-formed*, i.e., there exists a common environment in which they work

Issues in composition:
– control: error states (outputs are not acceptable inputs for the other)
– timing: time errors (one component cannot let time pass)

Game-theoretic view: interface compatibility checking using algorithms for solving timed games

Specific case:
– Timed interface automata with *input* and *output* invariants

Timed I/O Automata

[Kaynar & Lynch, 2003/2004]

Timed I/O Automata have:

– set X of internal variables, defining set Q of states;

– internal (H), input (I) and output (O) actions

– discrete transitions and timed trajectories

Requirements:

– *input action enabling*: $\forall x \in Q\,; \forall a \in I\ \exists x' \in Q\ .\ x \xrightarrow{a} x'$

– *time passage enabling*: in every state, time can either reach infinity or there is a trajectory which is (right-)closed and has a controllable action ($H \cup O$) enabled in its last state

Two TIOA are *comparable* if they have the same external actions.

Two TIOA are *composable* if they have disjoint internal variables and outputs, and hidden actions of one are not actions of the other.

Implementation relation $\leq$ is *trace inclusion*.

Assume-Guarantee for Timed I/O Automata

1) $A_1||B_2 \leq A_2||B_2$ and $A_2||B_1 \leq A_2||B_2$ imply $A_1||B_1 \leq A_2||B_2$ if:
– traces of A_2 and B_2 are closed under limits (*safety* properties)
– traces of A_2 and B_2 are closed under time extension
(do not impose stronger time passage constraints than $A_1||B_1$)

2) Conditions on A_2 and B_2 can be relaxed by introducing variant contexts A_3 and B_3, closed under limits and time-extension. Then:
$A_2||B_3 \leq A_3||B_3$ and $A_3||B_2 \leq A_3||B_3$ and
$A_1||B_3 \leq A_2||B_3$ and $A_3||B_1 \leq A_3||B_2$ imply $A_1||B_1 \leq A_2||B_2$

Reasoning can be extended to *liveness* (with more complex conditions)

Problems in compositionality

Composability of components

– typically (in timed $\{$automata, diagrams, I/O automata$\}$): a separate precondition to any assume-guarantee rule

– timed interfaces: optimistic composability view (context *exists*)

– more general frameworks for composability (urgency types, etc.)

Q: what restrictions result in simple composability check?

Safety and Liveness

– most assume-guarantee results concerned with safety

– liveness in a timed context – for timed I/O automata

Q: how to extend liveness results for other models ?

Problems in compositionality (cont.)

Completeness of assume-guarantee methods

– reasoning is usually incomplete for liveness; sometimes for safety

– [Namjoshi, Trefler 2000] give complete rule in untimed setting

– [Maier 2003]: cases where assume-guarantee cannot be both sound and complete

Q: in which setting is there completeness ? usable in practice ?

Automation of assume-guarantee checking

Q: for given goal $P_1||Q_1 \models S$, how to split $S = P_2||Q_2$?

Q: if helper assertions/contexts are needed, how to generate them ?

– some answers (w/o explicit timing) in [Namjoshi, Trefler 2000]

Generating abstractions for timed systems

– related to question of generating appropriate environments