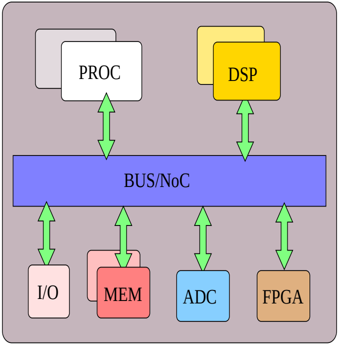
Incremental Design Process, Verification of Hardware Components and Abstraction Method for the Verification of System on Chip

Cécile Braunstein

University of Paris 6 (UPMC/LIP6/SOC)

Paris, May 14th 2007

Design of System on Chip



System on Chip (Soc)

Set of components needed to perform a complex function
Design of a SoC :

- Re-use of IP's (TLM/RTL)
- IP may be at different abstraction levels (Algorithm/RTL/NetList/transistors)

IP cores

- Come from different academic University or industrial company
- Must be easily adaptable and well specified
- Must be evolutive

Validation of System on Chip

Two main issues

1. Verification of each IP's
2. Verification of composed system

Validation Methods

- Simulation techniques : non-exhaustive, time expensive
- Formal Verification : exhaustive, memory expensive

Formal Verification Methods

	automatic	size limited	exhaustive
Theorem proving	NO	NO	YES
Equivalence Checking	YES	YES	YES
Symbolic Model checking	YES	YES	YES
Bounded Model checking	YES	YES	NO

Our Contribution

Our goal

- Help designers for the design and the Verification of each IP's
  Create a strong link between design and verification process

- Method for the verification of Composed system
  Fight the state explosion problem

Our context

- Modeling level : Synchronous Moore Machine,RTL
- Logic specification : CTL logic
- Formal Method : Symbolic Model Checker

Outline

1. Incremental Design and Verification process for one component
   - The incremental design process
   - Definition of the increment
   - Consequence on the specification of the incremented structure
2. Abstraction Method for the Verification of a Composed System
   - Abstract Kripke Structure
   - From CTL to an Abstract Kripke Structure
   - Properties of the abstract component
3. Case study : Platform with VCI-PI protocol converter
4. Concluding Remarks and Perspectives

Incremental Design and Verification process for one component

A design and verification framework

- Respects a design methods based on successive additions of new behaviors
- Relates design process and verification methods by model checking

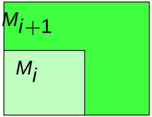
Design process per addition

- B Method [Abrial 85] : successive refinements, no new behavior
- Feature Integration [Plath/Ryan 99] : no guarantee of non-regression

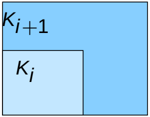
The incremental design process

A design framework inspired by hardware designers :

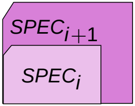
- Successive additions of new behaviors
- Conservation of existing behaviors : non-regression guarantee



Moore machine



Kripke structure



CTL formulas

In a general case when K_{i+1} simulates K_i

- ACTL Property preservation $K_{i+1} \Rightarrow K_i$ [Grumberg/Long 91]
- ECTL Property preservation $K_i \Rightarrow K_{i+1}$ [Loiseau and al. 95]

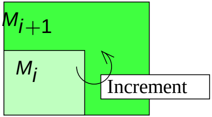
Incremental design

- CTL Property transformation $K_i \Leftrightarrow K_{i+1}$ [Braunstein/Encrenaz 04]

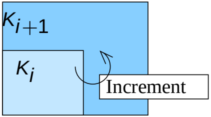
The incremental design process

A design framework inspired by hardware designers :

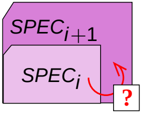
- Successive additions of new behaviors
- Conservation of existing behaviors : non-regression guarantee



Moore machine



Kripke structure



CTL formulas

In a general case when K_{i+1} simulates K_i

- ACTL Property preservation $K_{i+1} \Rightarrow K_i$ [Grumberg/Long 91]
- ECTL Property preservation $K_i \Rightarrow K_{i+1}$ [Loiseau and al. 95]

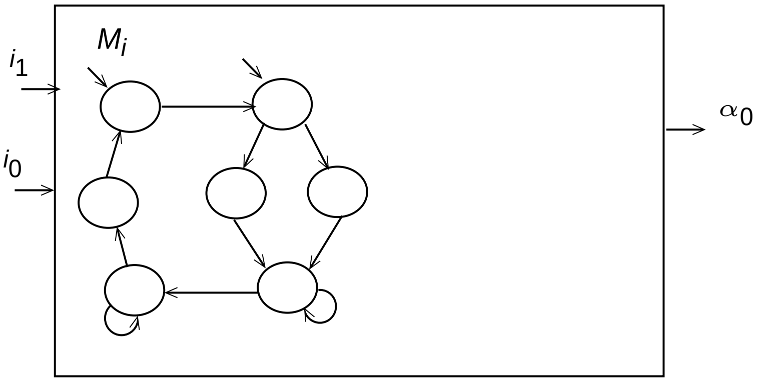
Incremental design

- CTL Property transformation $K_i \Leftrightarrow K_{i+1}$ [Braunstein/Encrenaz 04]

Definition of the Increment

Increment INC reacts to a set of new events at the interface

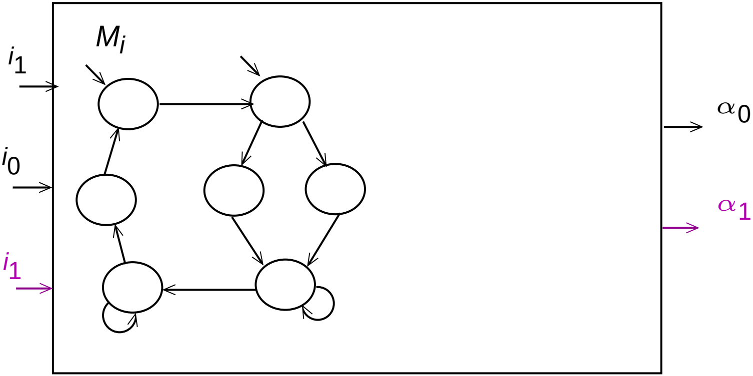
- Each event has quiet values and active values
- No new initial state, No behavior overriding
- M_{i+1} simulates M_i



Definition of the Increment

Increment INC reacts to a set of new events at the interface

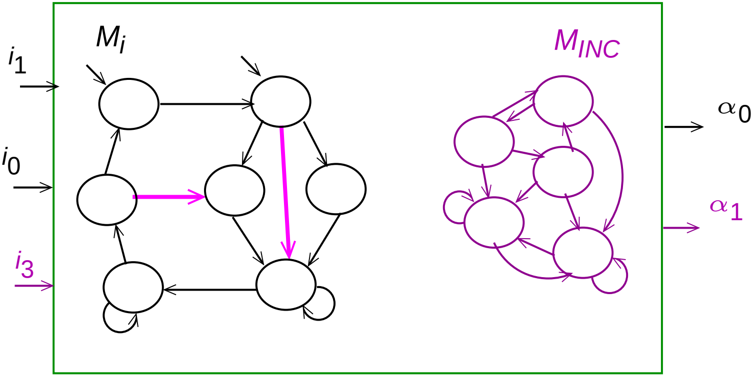
- Each event has quiet values and active values
- No new initial state, No behavior overriding
- M_{i+1} simulates M_i



Definition of the Increment

Increment INC reacts to a set of new events at the interface

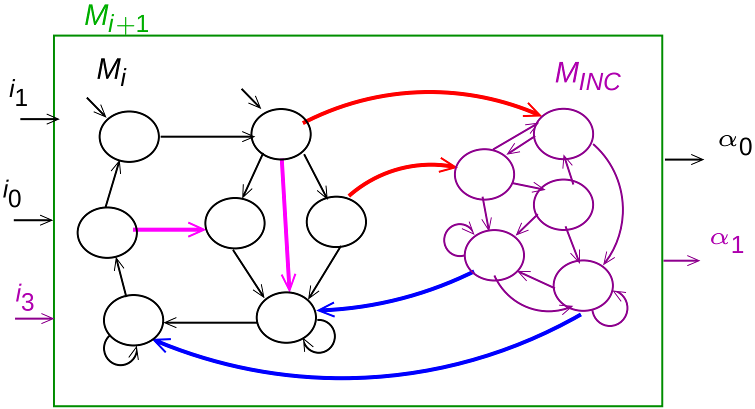
- Each event has quiet values and active values
- No new initial state, No behavior overriding
- M_{i+1} simulates M_i



Definition of the Increment

Increment INC reacts to a set of new events at the interface

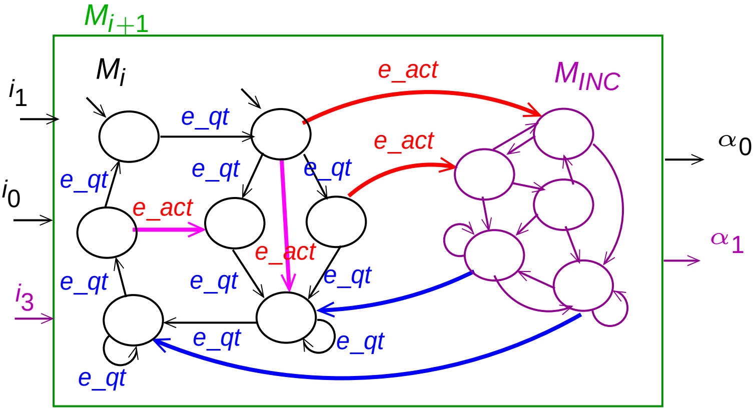
- Each event has quiet values and active values
- No new initial state, No behavior overriding
- M_{i+1} simulates M_i



Definition of the Increment

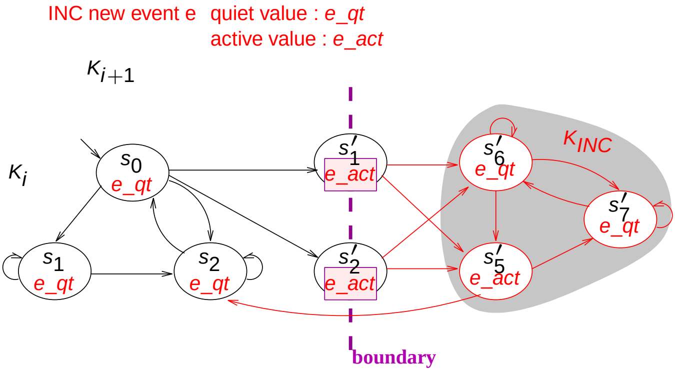
Increment INC reacts to a set of new events at the interface

- Each event has quiet values and active values
- No new initial state, No behavior overriding
- M_{i+1} simulates M_i



Incremented structure K_{i+1}

Boundary between the old and the new behaviors



Consequence on the specification of K_{i+1}

Incorporation the component's specification

- Can you derive the specification of K_i into a part of the specification of K_{i+1} ?
- Let φ be such that $K_i, s_0 \models \varphi$, what is φ' such that :

$$K_i, s_0 \models \varphi \Leftrightarrow K_{i+1}, s_0' \models \varphi'$$

Incorporation the increment's specification

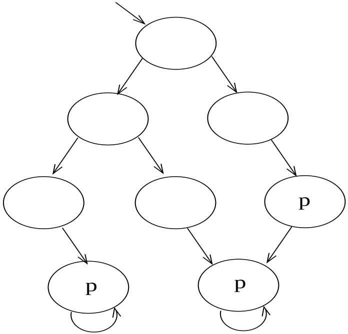
- Can you derive the specification of K_{INC} into a part of the specification of K_{i+1} ?
- If we can perform this we can obtain a big part specification of K_{i+1} from K_i and from K_{INC} !

Incorporation of the component's specification

$K_i \models AFp$

new even : e ; Quiet value : e_qt ; Active value : e_act

➢ $Ki + 1 \models AF(p \vee e\_act)$

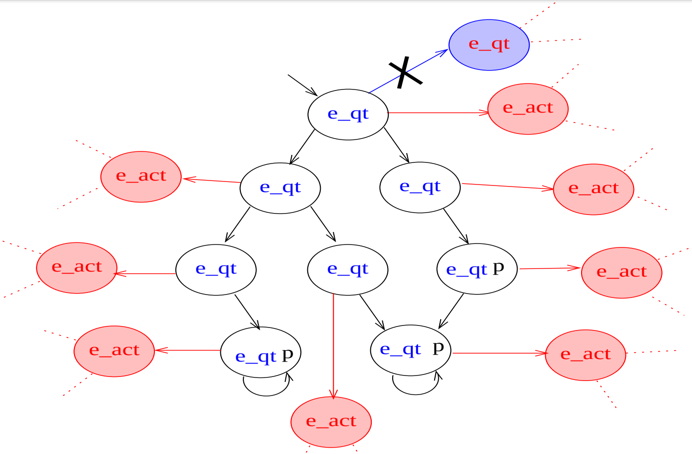


Incorporation of the component’s specification

$K_i \models AFp$

new even : *e* ; Quiet value : *e_qt* ; Active value : *e_act*

➢ $Ki + 1 \models AF(p \vee e\_act)$



Incorporation of the component's specification

Increment and Transformation rules [AVOCS 04,STTT 07]

- All CTL operators are transformable from K_i to K_{i+1}
- All CTL formulas are transformable by recursively applying the transformation from K_i to K_{i+1}
- Same complexity of the transformed CTL formulas
- Proves the non-regression of the specification by construction

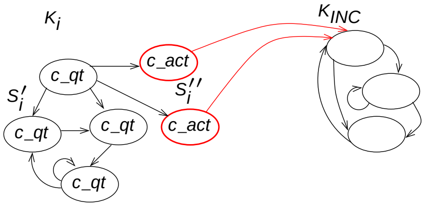
Application

- A concrete component design (VCI-PI protocol converter)

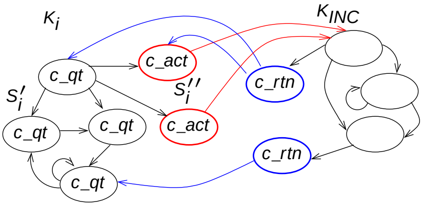
But we do not take into account the added behaviors !

Can you derive the specification of K_{INC} into the specification of K_{i+1} ?

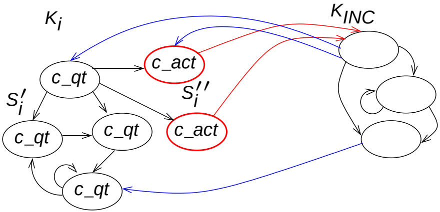
Return Boundary



(a) Without return



(b) With a special return value



(c) Without special return value

Incorporation of the increment's specification [LPAR 06]

(a) The specification of K_{INC} holds in K_{i+1} as soon as the active value holds

$$K_{INC} \models \varphi \Rightarrow K_{i+1} \models \mathbf{A}(e\_qt\mathbf{W}(e\_act \wedge \mathbf{AX}\varphi))$$

(b) The specification of K_{INC} holds in K_{i+1}as soon as the active value holds and until the occurrence of a return value

$$K_{INC} \models \varphi \Rightarrow K_{i+1} \models \mathbf{A}(e\_qt\mathbf{W}(e\_act \wedge \mathbf{AX}\boxed{\varphi'}))$$

(c) Not enough characterization of the return value but the "non-regression" rules still hold.

Conclusion on the Incremental Design Process

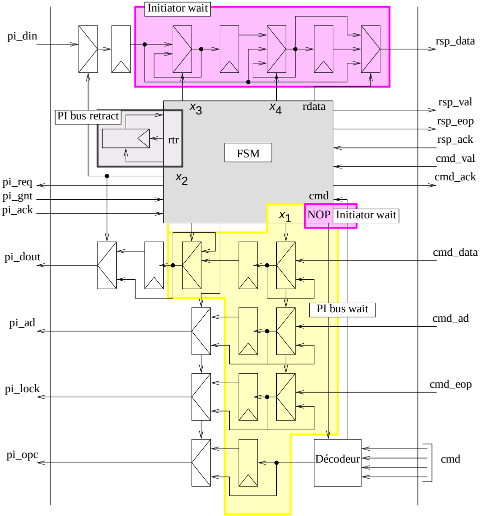
Results on the incremental design process

- Automatic transformations of component specification
- Automatic transformations of increment specification
- Particularized increments to the control flow of pipeline
- Automatic transformations of the stuttering increment [RSP 06]
- Specification of K_{i+1} guarantied by construction
- Application to a concrete component design and verification (VCI-PI wrapper)

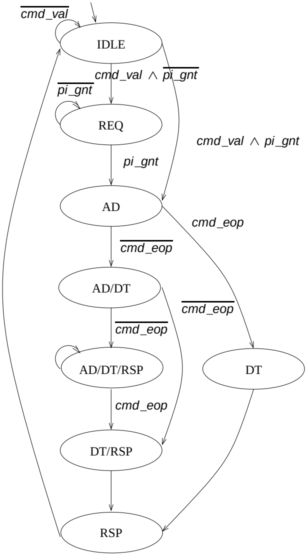
Ongoing work

- Tool for automatic integration of increment
- Use of specification as component abstraction in a compositional verification flow

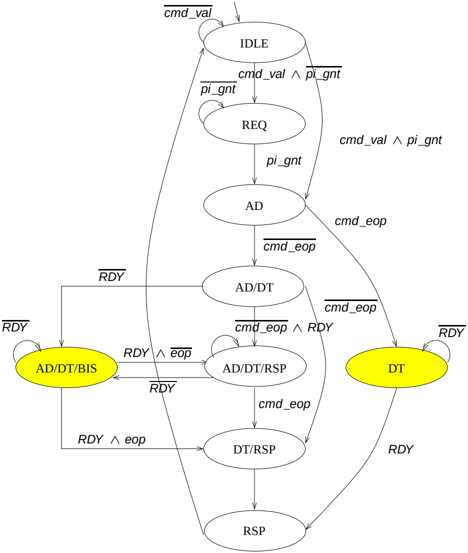
Design of VCI-PI master wrappers - Data Path



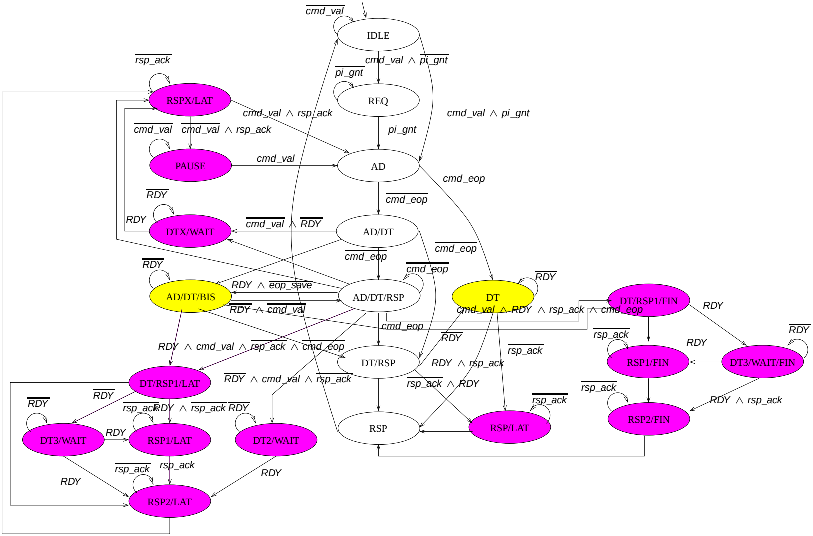
Design of VCI-PI master wrappers - Control Part



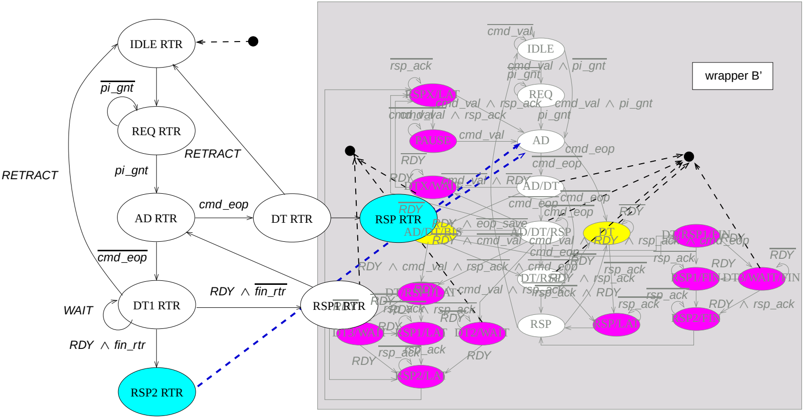
Design of VCI-PI master wrappers - Control Part



Design of VCI-PI master wrappers - Control Part



Design of VCI-PI master wrappers - Control Part



Abstraction Method for the Verification of a Composed System

Verification of Composed Components

- SoC : set of interacting IP's
- Each IP verifies its specification (CTL)

Goal : Check a global property on the product of components

Verification by Model Checking

Impossible for the complete system : state-space explosion

➢ We need abstraction !

Abstraction

Remove some information on the concrete system Σ in order to obtain a smaller abstract system Σ_A.

Abstraction Methods

An abstraction should be :

- Sound : $\Sigma_A \models \varphi \implies \Sigma \models \varphi$
- Effective : manageable by model checker tools
- Automatic : does not negate the automatic method of verification

Over-approximation Abstraction

- Simulation relation between the concrete and the abstract model
- Preservation of ACTL and LTL logics

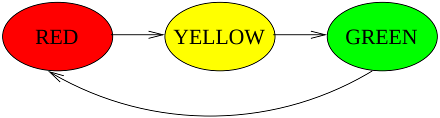
Abstraction Methods

- Cluster concrete states into abstract states (Data abstraction [Long 93], Predicate abstraction [Graf/Saidi 97])

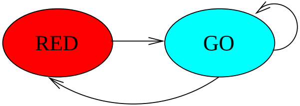
Abstraction Counter-example

An abstraction is less precise

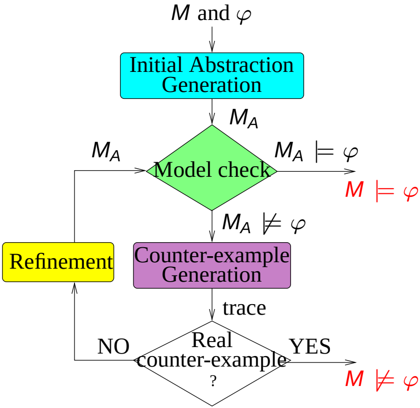
$M \models AGAF(state = red)$



$M_A \not\models AGAF(state = red)$



Counter-example guided abstraction refinement loop (CEGAR [Clarke/Grumberg and al. 00])



Model checker with CEGAR

- Software verification
  - SLAM [Ball/Rajamani 02]
  - BLAST [Henzinger and al. 03]
  - YASM [Gurfinkel and al. 06]
- Hardware Verification :
  - VCEGAR [Kroening and al. 07]

Our Abstraction Approach

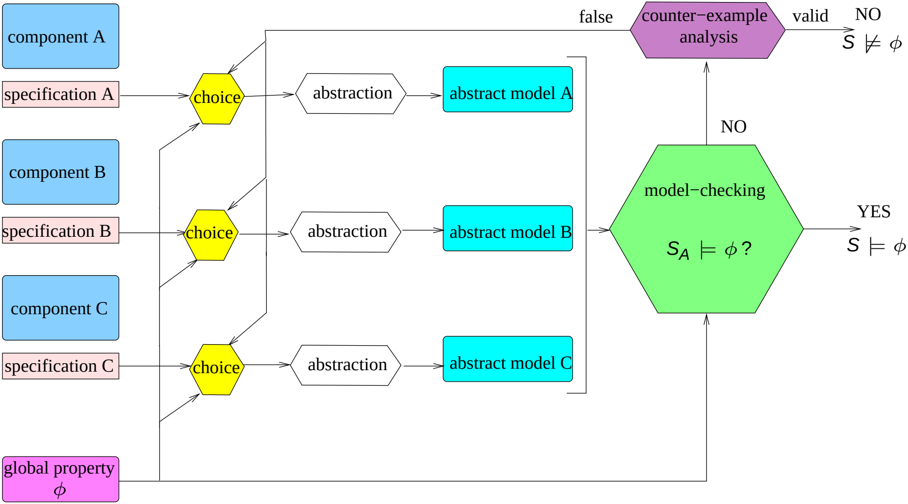
Our Abstraction

- Build the abstraction directly from the specification of each IP's
- Build a conservative abstraction that preserves ACTL formulas
- Integrate the framework in CEGAR loop

Related Works

- Abstraction from Specification
    - Verified systems from Verified components (LTL) [Xie/Browne 03]
    - Cando Object (PSL) [Schickel/Eveking 06]
- Environment Abstraction
    - Tableau construction (ACTL) [Peng/Tahar 02]
- Property synthesis
    - Monitor (PSL) [Morin-Allory/Borrione 06]
    - Prosyd project (PSL) [Bloem/Pnueli 05]

Our framework



Abstract Kripke Structure (AKS)

Definition

An *Abstract Kripke Structure* (AKS) is a tuple $K_A = \langle AP, S, S_0, \mathcal{L}, R, F\rangle$.
where the labeling function : $\mathcal{L} : S \rightarrow 2^{Lit}$.
$Lit = AP \cup \{\overline{p} \mid p \in AP\}$

Knowledge of an atomic proposition *p* in a state *s*

$p \notin \mathcal{L}(s) \wedge \overline{p} \notin \mathcal{L}(s)$	p is don't care in s
$p \notin \mathcal{L}(s) \wedge \overline{p} \in \mathcal{L}(s)$	p is false in s
$p \in \mathcal{L}(s) \wedge \overline{p} \notin \mathcal{L}(s)$	p is true in s
$p \in \mathcal{L}(s) \wedge \overline{p} \in \mathcal{L}(s)$	p is inconsistent in s

From CTL to an Abstract Kripke Structure

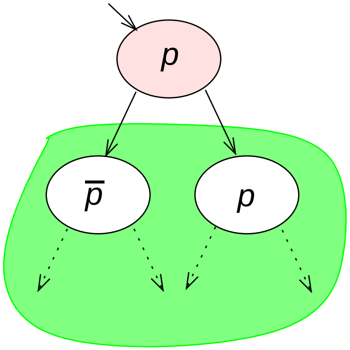
Construction of an AKS

- Build the less constrained structure K_φ where φ holds.
- Obtain an abstraction directly from a syntactic analysis of the formulas
- Have many atomic propositions with a *don't care* value

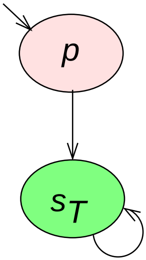
From CTL to an Abstract Kripke structure

$$\varphi = p$$

Concrete Kripke structure

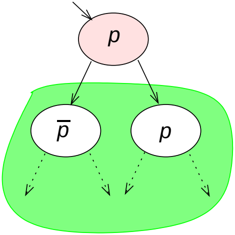


Abstract Kripke structure

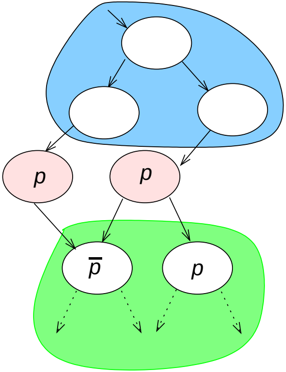


From CTL to an Abstract Kripke structure

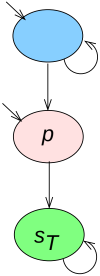
$$\varphi = AFp$$



Concrete Kripke structure

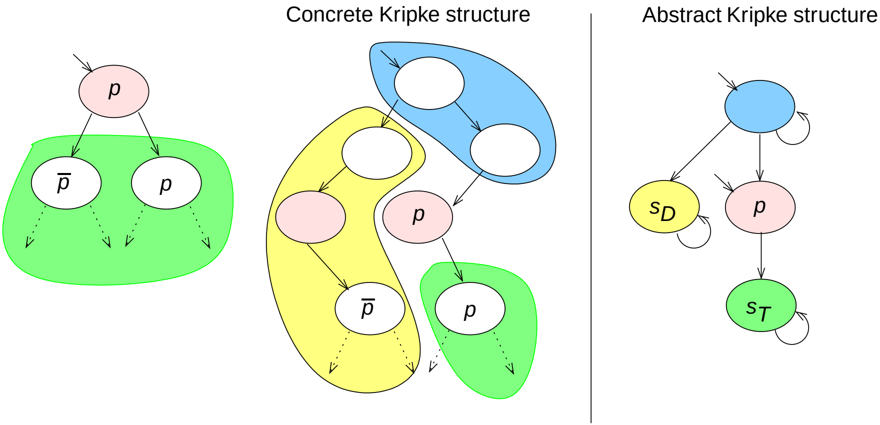


Abstract Kripke structure



From CTL to an Abstract Kripke structure

$$\varphi = EFp$$



Properties of the Abstract Component [ACSD 07]

We define an recursive algorithm to build AKS from all CTL$\backslash X$ formulas φ

All AKS K_φ obtained from our algorithm are such that :

- $K_\varphi \models \varphi$
- For all $K \models \varphi$ there exists a simulation relation such that $K \preceq K_\varphi$

Composition of AKS

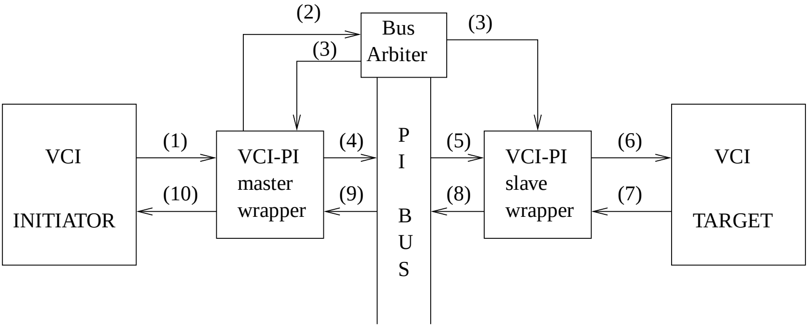
By assume-guarantee reasoning, the composition of AKS Σ_A is an abstraction for a concrete component Σ and $\Sigma \preceq \Sigma_A$.

➢ ACTL formulas are preserved from Σ_A to Σ

Abstraction characteristics

- Many states with don't care atomic propositions
- Depth of the abstract Kripke structure is small

Case study : Platform with VCI-PI protocol converter



Global Property to check

AG(initiator[i]_state = TRANS $\Rightarrow$ **AF** (target[j]_signal_rsp = 1))

Experiments with the SMC VIS

The first experiments are performed by hand writing refinement steps
The property takes 4 refinement iterations (10 formulas)

Platform name	FSM depth	# BDD var	BDD size (# nodes)	# Reach. states	Reach. time	Checking time
Concrete 1 master 1 slave	475	289	34 578	8.50E+06	51s	6.70s
Abstract model	10	302	1 002	1.69E+22	8s	0.25s
Concrete 2 masters 1 slave	604	436	161 846	3.10E+10	40min	48min
Abstract model	14	501	1 564	2.50E+26	12s	8.25s

Experiments realized on a Pentium IV 3,2GHz with 2GB ram.

Concluding remarks

Verification of one component

Incremental Design process

- Formalization of a framework close to hardware designers
- Automatic evolution of the specification during the design process

Application to the design and the verification of wrappers VCI-PI master and slave.

Verification of composed component

Abstraction method directly from the specification of each component

- Reduction of the explored tree depth
- Reduction of the number of BDD nodes

Application to the verification of platform composed of wrappers VCI-PI master and slave.

Perspectives

Abstract counter-example analysis

- How to detect a "spurious" counter-example over the abstract model ?
- How to get feed back from a counter-example in order to produce a stronger set of properties ?
- How to add properties if the specification is not complete enough ?

Perspectives