# Specification and Verification of Inter-Component Constraints in CTL

Nguyen Truong Thang Takuya Katayama
Japan Advanced Institute of Science and Technology – JAIST
{thang, katayama}@jaist.ac.jp

# Contents

# CB Software (1/4)

199x: components with talking features only

- Component-based software: structured from a set components
  - Ideally, components are plug-and-play.
  - Flexible for changes: handling new functional requirements or operating platforms.
  - E.g.: mobile phones → camera-equipped mobile phones.

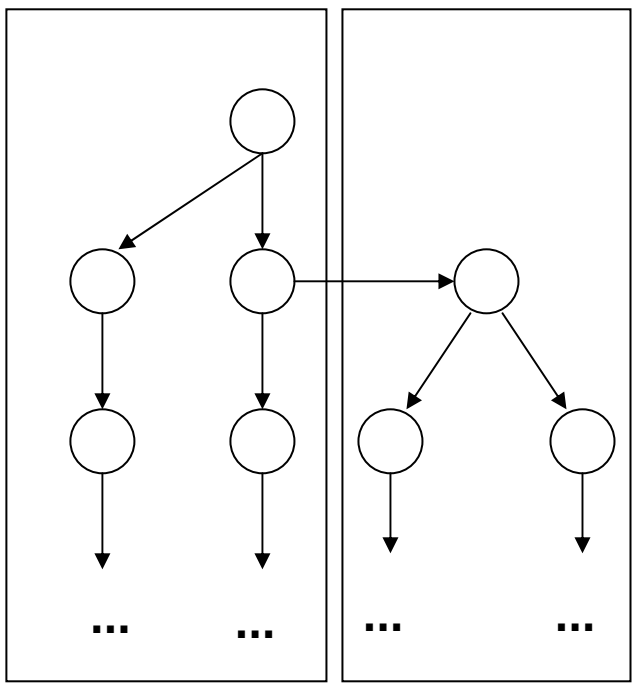
evolving

- Many current software practice are essentially component-based.
  - Feature-oriented software.
  - Each feature is treated as a large component which is formed from several member components.

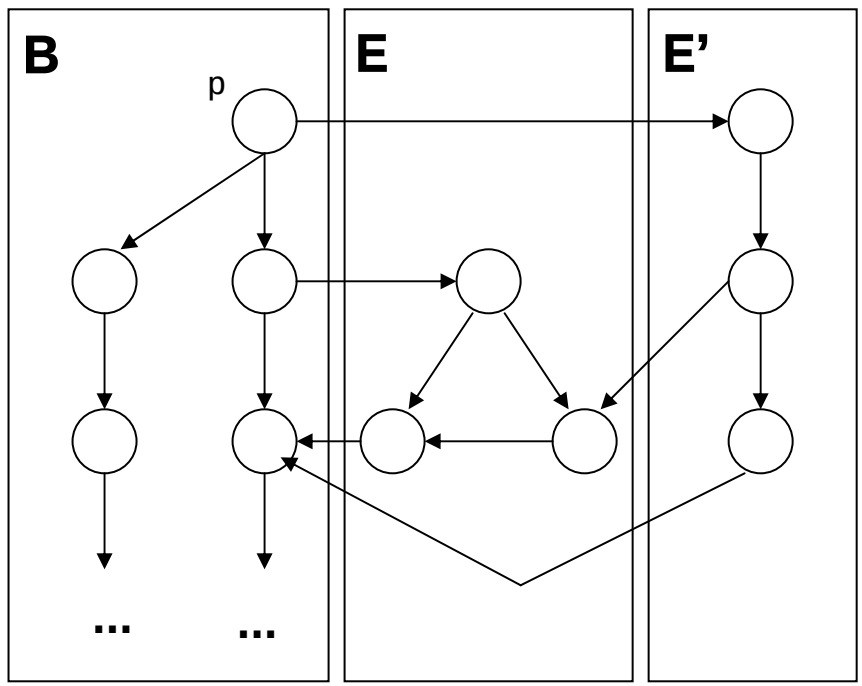
2004: offering varieties of features via extra components:
- Email/MMS
- Photo-shooting
- Contact-less IC
- Web browsing
- Document Viewer
- GPS etc.

# Component-Based Software (2/4)

- Components:
  - Component-Off-The-Shelf (COTS): independent components in which computation paths rarely interleave each other (only a single *exit* state, no *reentry* state).
  - Component refinement: interleave at some degree.
  - This work: focusing on refinement (also applicable to COTS). Specifically, a property initially holds in B. How to verify that subsequent refinements like E and E' still preserve p in the composition component.



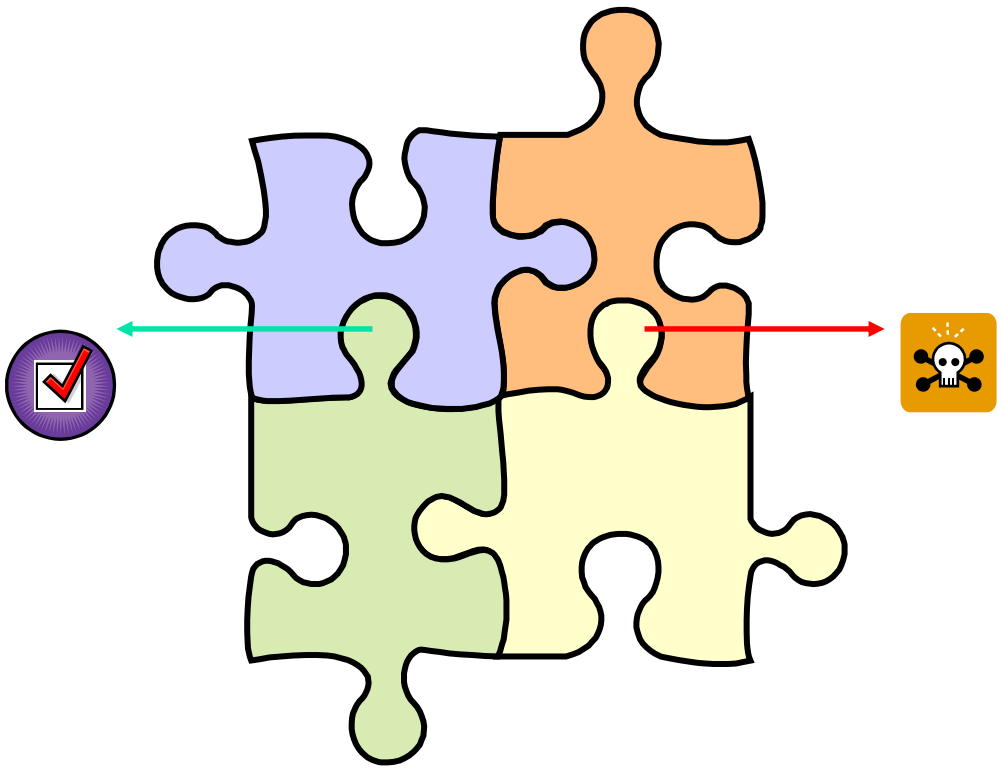

**COTS**




**Refinement**

# Specification and Verification of CBS (3/4)

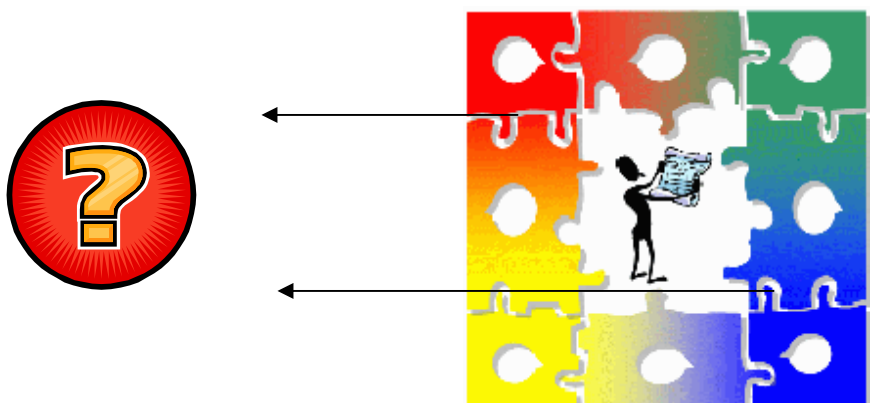
- Current practice in component technology:
  - Component plugging: only up to the level of syntactical matching.
  - The issue: after being plugged, the components are ***inconsistent*** with each other.
  - The work: focusing on the consistency in terms of CTL properties.



# Specification and Verification of CBS (4/4)

- An important issue of component-based software paradigm:
  - Specifically, what to formally specify component consistency and how to verify it in ***consistent*** and ***efficient*** manner?

- Solution:
  - Component specification: enforced with interface-mapping compatibilities and consistency constraints.
  - Verification: via Open Incremental Model Checking (OIMC).
  - OIMC: using assumption at *reentry* states, checking if the *preservation constraints* are preserved at the interface between components. If so, the consistency among components is guaranteed. (explained later in Software Verification section)



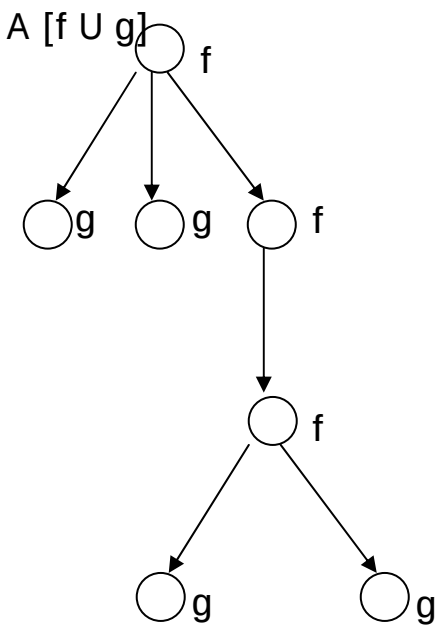
# Contents

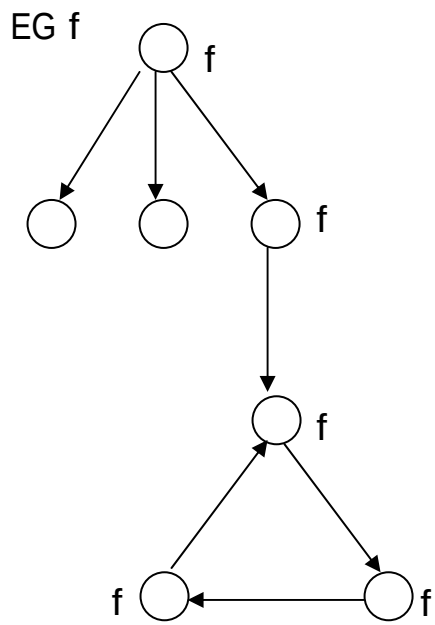
- Introduction
- Software Verification
  - Model Checking: CTL and Assumption Model Checking
  - Incremental Model Checking
- A Formal Model of Components
- Incremental Verification of Component Consistency
- Component Specification and Verification
- Conclusion

# Model Checking & CTL (1/4)

- CTL* logic: constructed from two quantifiers:
  - A (for all paths) and E (for some path); plus five temporal operators: X (next), F (eventually), G (always), U (until), R (release).

- CTL: a true subset of CTL*.
  - 10 basic normal CTL properties: AX f, EX f, AF f, EF f, AG f, EG f, A [f U g], E [f U g], A [f R g], E [f R g]; where f and g are CTL or atomic propositions.



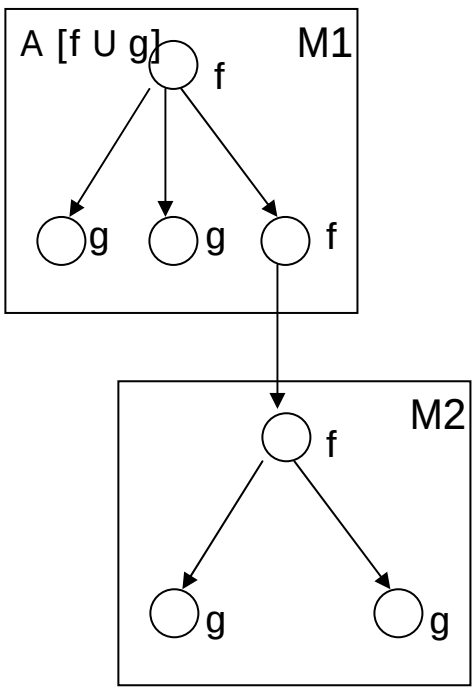
# Model Checking & CTL (2/4)

- Definition: The *closure set*, cl(p), of p is the set of all sub-formulae of p.
  - p is an atomic proposition: $cl(p) = \{p\}$
  - p is among AX f, EX f, AG f, EG f, AF f, EG f: $cl(p) = \{p\} \cup cl(f)$
  - p is among A [f U g], E [f U g], A [f R g], E [f R g]: $cl(p) = \{p\} \cup cl(f) \cup cl(g)$
  - $p = \neg f$: $cl(p) = cl(f)$
  - $p = f \wedge g$ or $p = f \vee g$: $cl(p) = cl(f) \cup cl(g)$

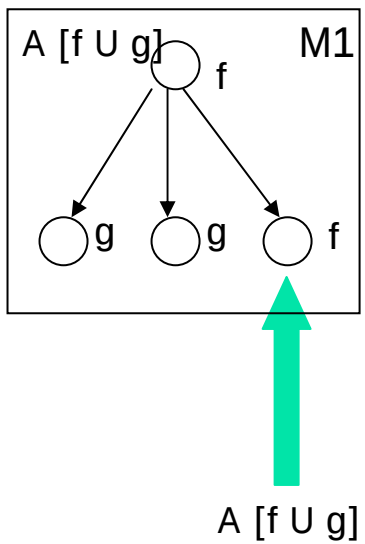
- In model checking, the characteristic is inside-out.
  - To verify p in a model M, all sub-formulae closure set cl(p) of p are in general checked on the way.

# Assumption Model Checking (3/4)

- Idea [Laster98]: 2 sequential modules M1, M2.
  - Possible to model check within M1 only if knowing the labels at the interface states between M1 and M2 by representing the whole M2 with those labels.
  - A critical note on AMC: There should be no circle involving the interface nodes of M1 and M2.




=



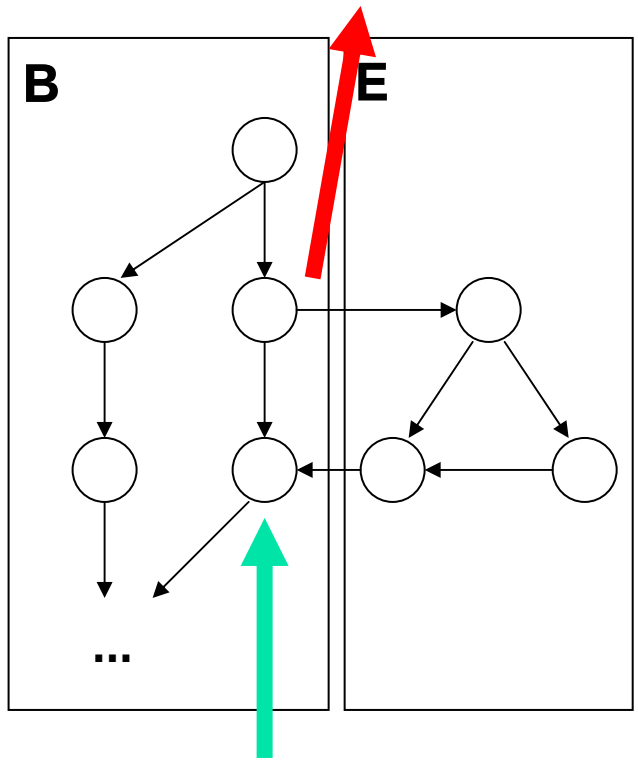

Assumption model checking is ***reliable*** only if the assumed labels at the interface states are ***proper***.

# Incremental Model Checking (4/4)

- Incremental verification (or Open Incremental Model Checking) [Fisler01 etc]:
  - An application of Assumption Model Checking.
  - Difference from AMC: ensuring the preservation of constraints.
  - Efficient: model checking each component separately.
  - Open: handling even unanticipated future changes.

- OIMC:
  - Focusing on component refinement, but also applicable to COTS.
  - Initially, a property under consideration holds in a base component.
  - The property is guaranteed to hold in the system as long as other components preserve constraints at the interface of the base component.

After assumption model checking in E, if the ***constraints*** at this state are ***preserved***, there is no need to check further in B.

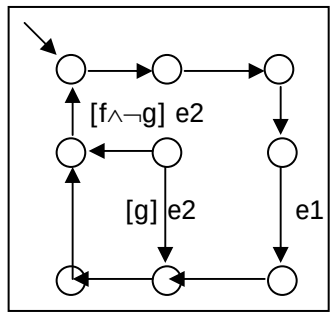



The assumption made at this state represents the computation tree in B

# Contents

# A Formal Model of Components (1/2)

- A component is formally represented by a state transition model:
  - A set of states: S
  - A set of input events: $\Sigma$
  - An initial state: $s_0$
  - A state transition function: R: S x PL($\Sigma$) $\rightarrow$ S
  - Labeling function at states: L – showing a set of atomic propositions to be true at a given state.



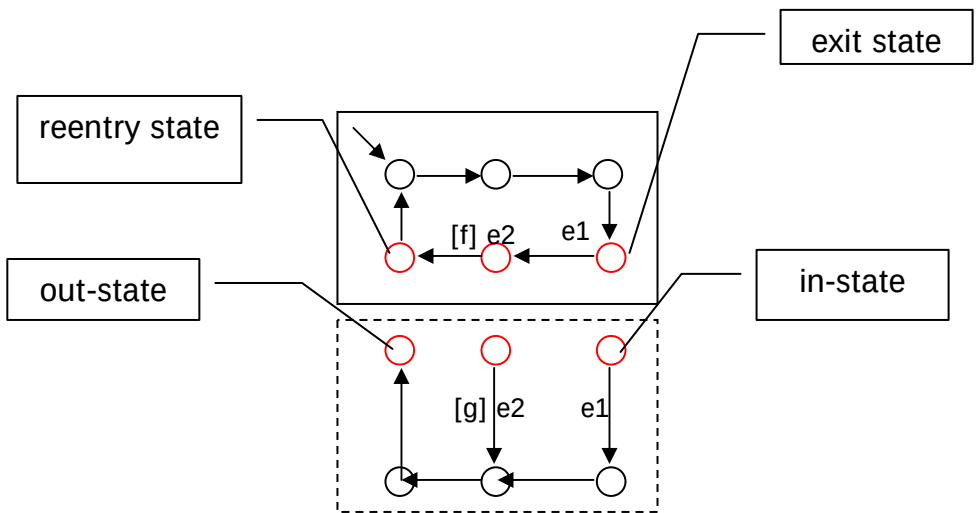

- Typical case: a base component B is extended with an extension component E.
  - B = <$S_B$, $\Sigma_B$, $s_{0B}$, $R_B$, $L_B$> (see figure in next page)
  - E = <$S_E$, $\Sigma_E$, $\perp$, $R_E$, $L_E$> ($\perp$ : no-care value)
  - B and E are either composite or primitive components.

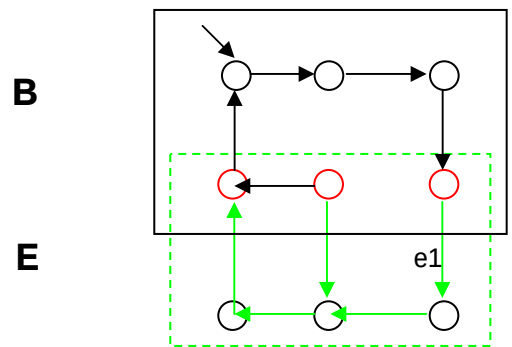
- Associated with a component is an interface of two state sets.
  - B: <exit, reentry> - at which control is released from/returned to the base.
  - E: <in, out> - states receiving/returning control respectively

# A Formal Model of Components (2/2)

- Interfaces of B and E to be mapped accordingly.
  - Defining the compatible conditions for which ex $\leftrightarrow$ i, o $\leftrightarrow$ re.
  - ex $\leftrightarrow$ i if $\wedge[L_B(ex)] \Rightarrow \wedge [L_E(i)]$, where $\wedge$ is the inter-junction.
  - o $\leftrightarrow$ re if $\wedge [L_E(o)] \Rightarrow \wedge [L_B(re)]$.

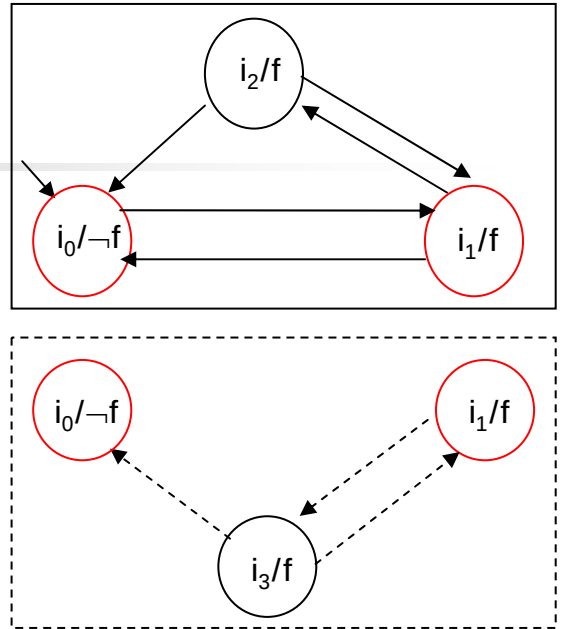
- The composition model C = $<S_C, \Sigma_C, s_{0B}, R_C, L_C>$ is defined via elements of B and E.
  - E can overrides some part of B.



# Contents

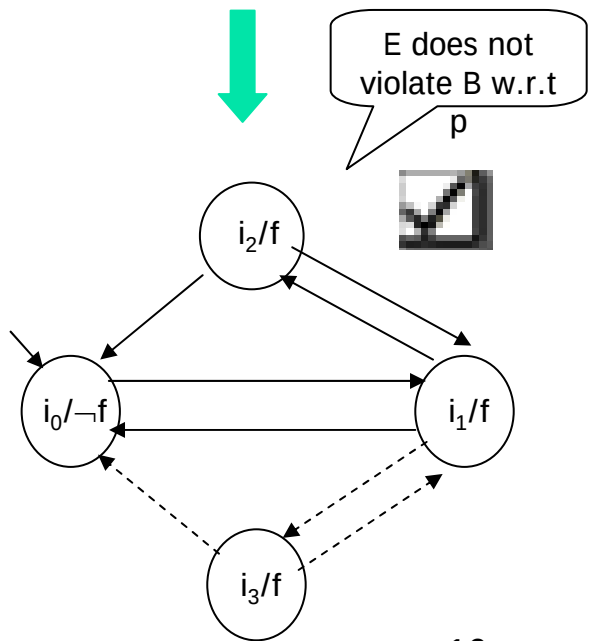
# Component Consistency (1/5)




- Definition: a property p holds on a model M = (S, $\Sigma$, $s_0$, R, L) if M, $s_0$ |= p.

- Component consistency definition:
  - In terms of CTL properties.
  - Initially, p holds on B = ($S_B$, $\Sigma_B$, $s_{0B}$, $R_B$, $L_B$), i.e. B, $s_{0B}$ |= p.
  - E does not violate p on B if within C, p still holds at $s_{0B}$ in C, i.e. C, $s_{0B}$ |= p.

- In the example: p = AG EX f
  - Initially: B, $i_0$ |= p
  - After composition, C, $i_0$ |= p
  - E does not violate p in B in this case.



# A Theorem on Component Consistency (2/5)

- Definition: Given a model M, the truth values of a state s with respect to a closure set cl(p), $V_M(s, cl(p))$ are
  - $\forall \phi \in cl(p)$: if M, s |= $\phi$ then $\phi \in V_M(s, cl(p))$.
  - Otherwise, $\neg\phi \in V_M(s, cl(p))$.

- Conformance condition: B and E conform with each other (with respect to cl(p)) at an exit state ex if $V_E(ex, cl(p)) = V_B(ex, cl(p))$.

- ***Theorem: Given a property p holding in B, E does not violate p in B if B and E conform with each other (w.r.t cl(p)) at all exit states.***
  - Regardless of composition type: additive or overriding.

# Incremental Verification of Components (3/5)

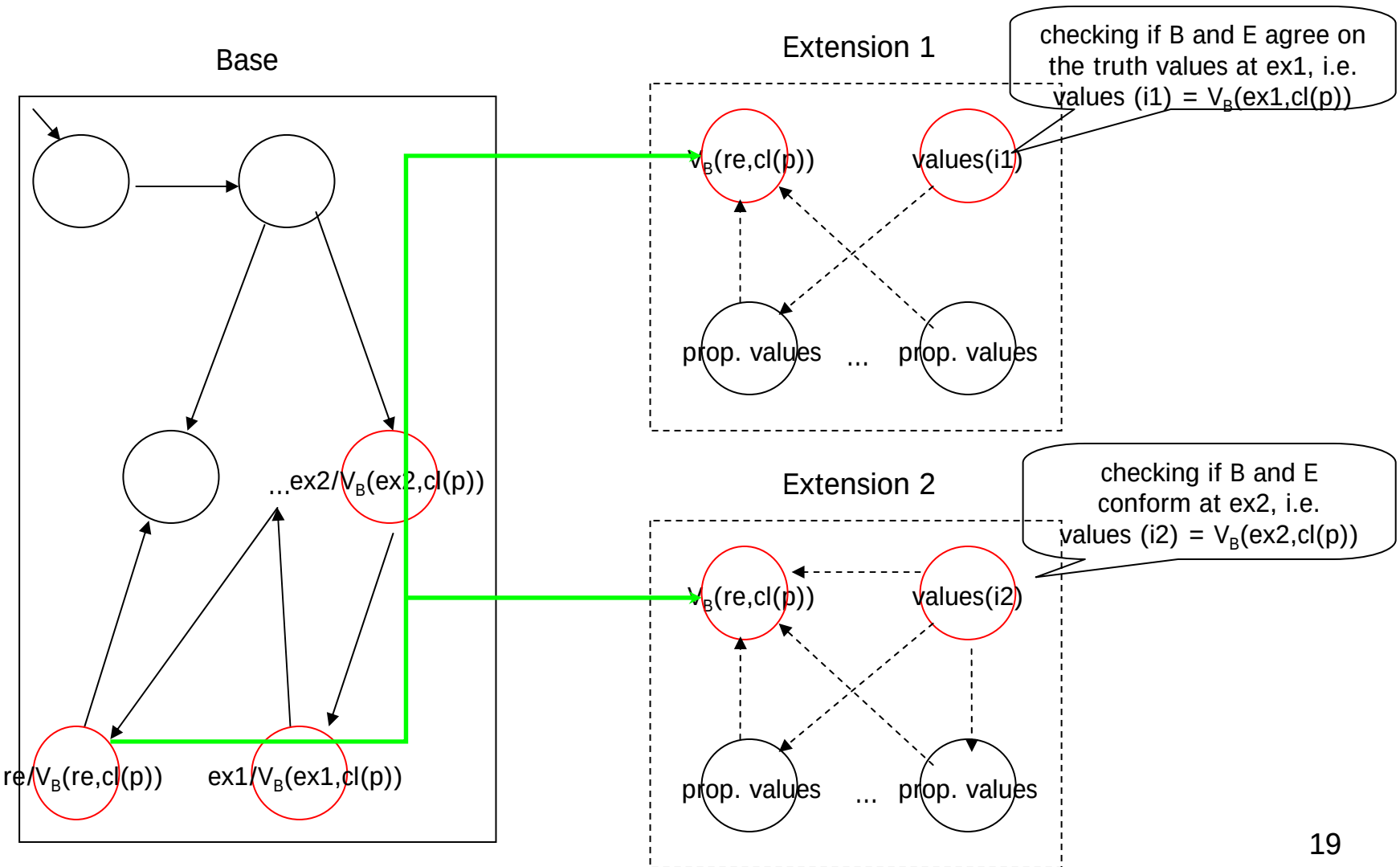
During verifying p on B, the preservation constraints pc(s) = $V_B$(s, cl(p)) at any interface state s are recorded.

**The algorithm of OIMC: within E (the refinement) only**

1. For each reentry state re in B, seeding pc(re), i.e. $V_B$(re, cl(p)), at the corresponding mapped out-state o in E.
2. For each in-state i in E: run the CTL assumption model checking procedure in E to check sub-formulae $\phi$, $\forall\phi \in$ cl(p).
3. Checking if $V_E$(i1,cl(p)), $V_E$(i2,cl(p)), ... are matched with the preservation constraints $V_B$(ex1,cl(p)), $V_B$(ex2,cl(p)), ... at respective mapped exit states ex1, ex2 ... of B.

Note: - In case of COTS, there is no assumption since no reentry states.
- Assumption model checking is then replaced by standard model checking.
- The constraints stay the same as above.

# Incremental verification of components (4/5)

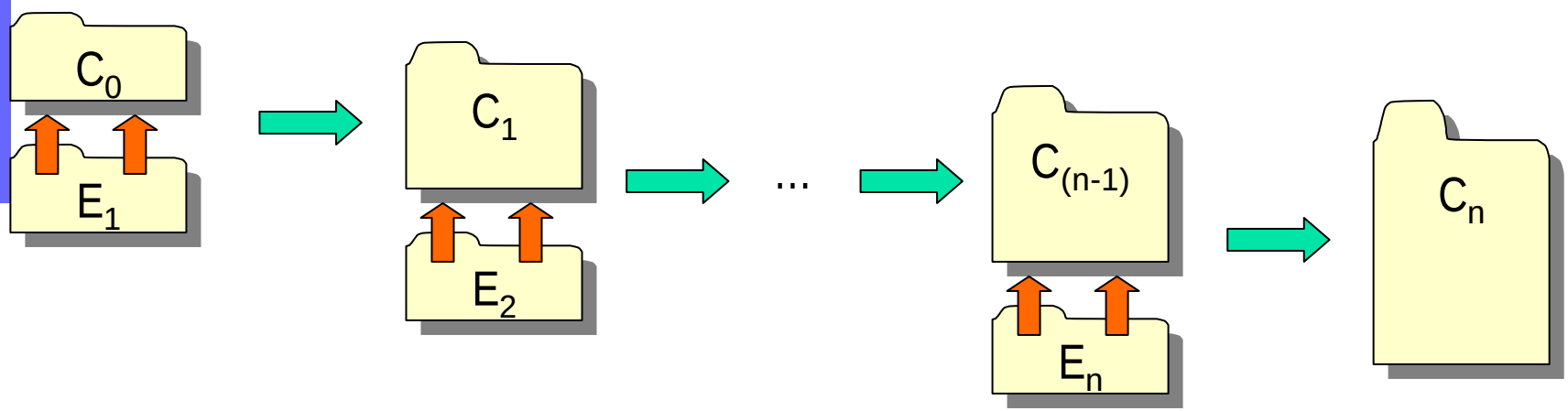


# Scalability of Incremental Verification (5/5)

- Considering subsequent component refinements.

- ***Theorem: The method preserves its incremental characteristic for any subsequent extensions as long as $E_i$ conform with $C_{(i-1)}$ at all exit states between them.***

  - The complexity only depends on the size of $E_n$ (extending the base $C_{(n-1)}$).




At all evolution steps, incremental verification for component consistency is scalable.

# Contents

# Component Specification (1/4)

- Component-based software:
  - Problem: components are often inconsistent after composition.
  - Consistency: several types $\Rightarrow$ focusing on CTL property preservation.
  - This work: enforcing component matching in terms of consistency semantic.

- Current component technology (OMG CORBA, Sun Java and JavaBeans, Microsoft .NET and COM/DCOM, UML/OCL etc): semantic is limited to a simple logic of weak expressiveness and syntactical component matching.
  - Internal to components.
  - Inter-component: Consumer.num_items $\leq$ Producer.num_items.
  - The underlying logic only expresses constraints at the moment the interface element is invoked, i.e. static view.

- Component specification:
  - Interface signature: traditionally, attributes and operations (static and syntactical matching).
  - Constraints: component matching in terms of semantic (via the interface compatibility in the formal model and the CTL consistency).

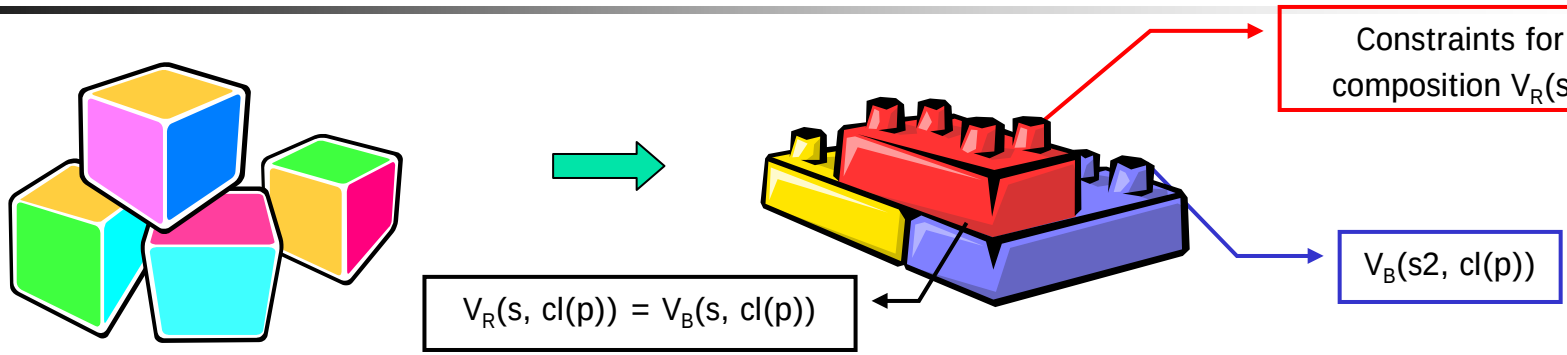
# Component Composition (2/4)

- Encapsulating temporal semantic constraints to component interface via 2 constraint types.
  - Plugging compatibility: for two components to be plugged together via exit-in and reentry-out states (p.11).
  - Consistency constraint: making components to be consistent after being composed (p.15).

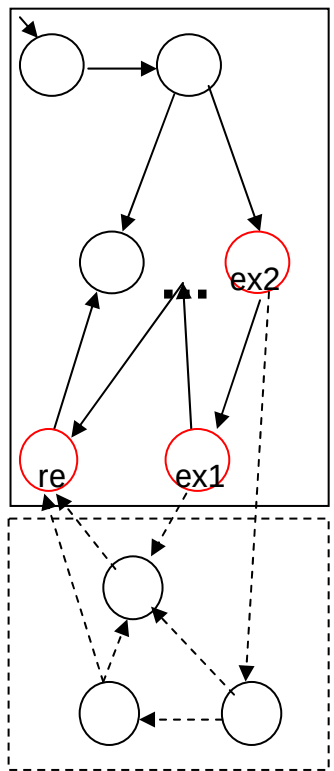
- Consistency constraint:
  - Written in CTL showing components' execution traces, i.e. dynamic view.
  - Regarding to a CTL property p inherent to a component B, at an interface state s, its constraints are $V_B(s, cl(p))$.

- Composing two components: C = B + E
  - Signature (attributes and operations): the sum of those from B and E.
  - Plugging constraint: taken as $L_B(s)$ or $L_E(s)$ accordingly.
  - Consistency constraint: taken as $V_B(s, cl(p))$ or $V_E(s, cl(p))$ accordingly.

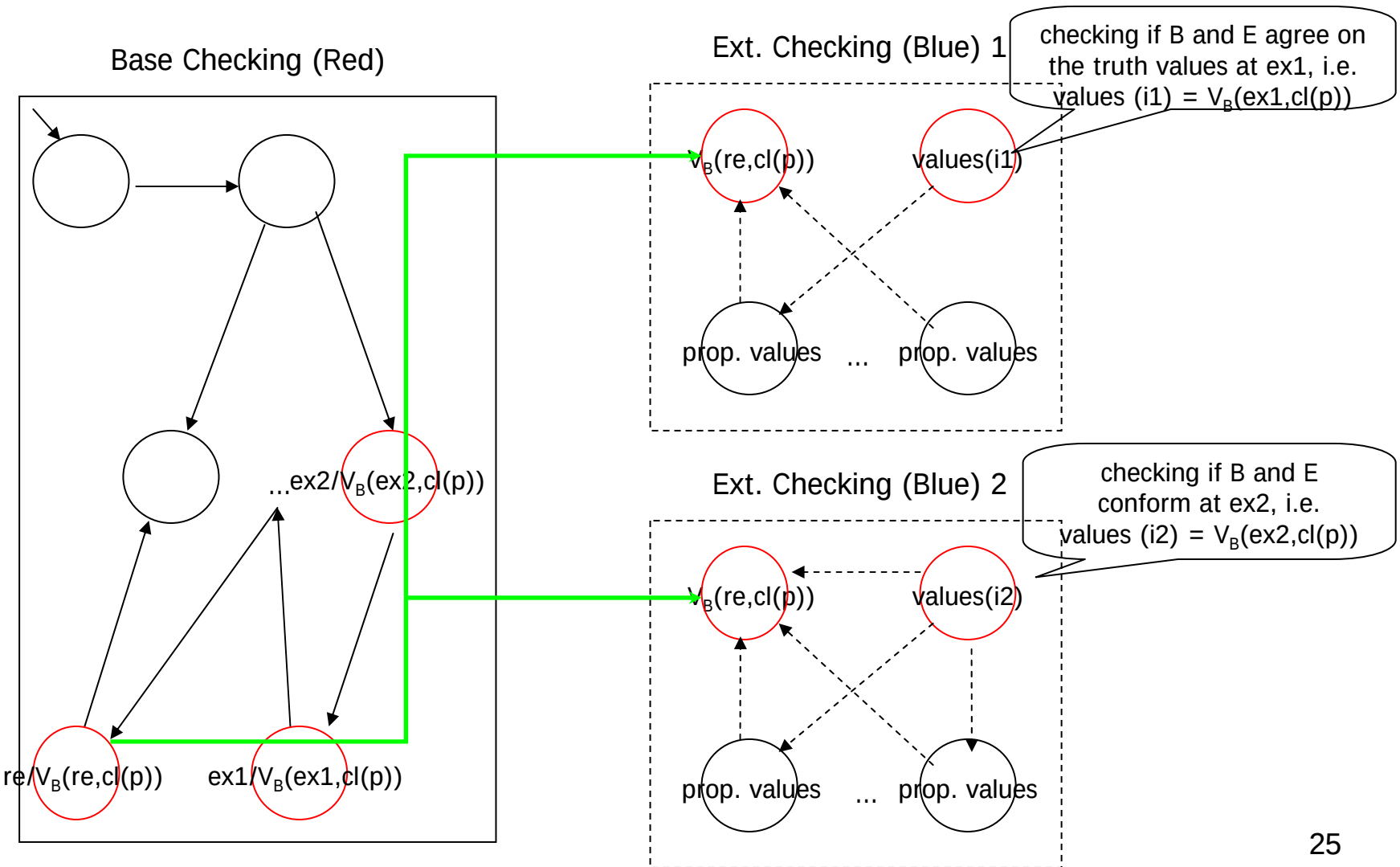
# Incremental Verification of Components (3/4)




- Initially, p holds in B.
- B (Red) and E (Blue) are composed via two exit states ex1, ex2 and a reentry state re.
- If $V_E$(ex1, cl(p)) = $V_B$(ex1, cl(p)) and $V_E$(ex2, cl(p)) = $V_B$(ex2, cl(p)), E does not violate p in B.
  - only executed in E (i.e. incrementally).
- Similar checking for Red and Yellow.



# Incremental Verification of Components (4/4)



# Contents

# Conclusion (1/3)

- Modular verification [Grumberg91, Kupferman98 etc]: based on assume-guarantee paradigm
  - Often dealing with hardware verification; modules are composed in parallel.
  - Verifying each module separately while assuming about the behaviors of the external environment and other modules.
  - Interfaces are pre-defined and static.
  - Verification task needs to re-run in the whole system if a new module is inserted or removed.

- Modular software verification [Laster98]: exactly Assumption Model Checking.
  - Characteristically different from hardware verification.
  - Taking the advantage of sequential nature in software.

- OIMC: the application of AMC in an open way (unanticipated future evolution via component refinement).
  - Open verification comes at the cost of fixed preservation constraints at interface.

# Conclusion (2/3)

- Open incremental model checking
  - Interface is dynamically defined.
  - Systems are more ***open*** for changes.
  - Only model checking within the new module, i.e. ***incremental***.
  - The approach is also scalable for the whole evolution process as long as bases and extensions pair-wise conform.

- Several research on software modules (components) compatibility or consistency.
  - Different types of consistency.

- [Chakrabarti02]: interface compatibility among software modules
  - Also state-based model.
  - Focusing on different aspects of component semantic: ***correctness and completeness of operation definitions within components***.
  - Complementary to the ***temporal consistency among components*** in this work.

# Conclusion (3/3)

- OIMC is based on assumption model checking.
  - AMC is not supported by well-known model checkers such as SMV, SPIN etc.

- Improvement from current stage:
  - Tool support: NuSMV2 is the target for adapting OIMC into real practice.

- NuSMV2 is selected as the target because:
  - open-source: comprehensibility and documentation.
  - derived, i.e. re-design and re-engineering, from SMV (focusing on CTL properties).