Specification and Verification of Inter-Component Constraints in CTL

Nguyen Truong Thang Takuya Katayama
School of Information Science
Japan Advanced Institute of Science and Technology
email: {thang, katayama}@jaist.ac.jp

ABSTRACT

The most challenging issue of component-based software is about component composition. Current component specification, in addition to the syntactic level, is very limited in dealing with semantic constraints. Even so, only static aspects of components are specified. This paper gives a formal approach to make component specification more comprehensive by including component semantic. Fundamentally, the component semantic is expressed via the powerful temporal logic CTL. There are two semantic aspects in the paper, component *dynamic behavior* and *consistency* - namely a component does not violate some property in another when composed. Based on the proposed semantic, components can be efficiently cross-checked for their consistency by an incremental verification method - OIMC, even for many future component extensions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

1. INTRODUCTION

As an unanimity within the software engineering community, high quality software are structured from lowly coupled *components*. Within the component-based approach, composing components properly is very essential. Component-based software idealizes the *plug-and-play* concept. The current component technology generally supports component matching at the syntactic level. Components can be syntactically checked and hence *plugged*. However, they do not *play* as expected. A major issue of concern is the mismatches of the components in the context of an assembled system. A main source of this phenomenon is because a component violates some property inherent to another. In our opinion, the problem is two-fold: the underlying logic is not powerful enough to express component properties; and even if formally specified, it is difficult to verify the properties in an open way - future components are not known in advance. For instance, temporal inter-component constraints are difficult to formally specify, much harder to check among components with the current specification methods. In this paper, the introduction of the temporal logic CTL [4] into component semantic is exactly towards that goal. This paper addresses two points: how to explicitly specify such a component semantic; and given that kind of information in the component interface, how to efficiently analyze components and to decide whether they are safe to be composed together.

Most current approaches for component interface definition deal with primarily syntactic issues among static interface elements such as *operations* and *attributes*, like those of the CORBA Interface Definition Language (IDL) [9]. Regarding a component's exact capability, essential semantic aspects of the component should also be described. In this paper, the *dynamic behavior* and *component consistency* are introduced, while the encapsulation principle is enforced. The dynamic behavior of a component is represented by a state transition model. Besides, associated with a component's behavior is a certain set of inherent properties. Certainly, another component, when interacting with that component, must preserve constraints at the interface of the former so that those inherent properties continue to hold. This characteristic is called *component consistency.* Moreover, as written in CTL, many complex semantic constraints of component consistency can be formally specified. The paper then presents an efficient algorithm to analyze consistency between components. Further, the algorithm is also scalable, not only the direct component extension but also many future compositions, as long as the consistency constraints at the interfaces are preserved.

In this paper, Section 3 introduces the formal dynamic behavior model of components. Section 4 is about component consistency and how to verify it. Later, Section 5 is concerned with specification of components and their composition.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2. BACKGROUND

The most common form of component deployment in practice, namely Commercial-Off-The-Shelf (COTS), is on very independent components. The computation paths of these components rarely interleave with each other. The relationship between COTS can be named *functional addition.* Besides COTS, there is another aspect of components involving in-house component development and integration. Components evolve through *functional refinement.* These

components are relatively coupled thus COTS is not a recommended option in this situation. There is a strong dependency from the refining component to the base component. Even though the discussion in this paper focuses on component refinement, the results can be well applied to COTS because analyzing COTS is obviously simpler. In practice, there is a trade-off between the simplicity of component specification and proper use of components. For COTS, component specification at the syntactical level may prove to be sufficient for component deployment in most of the cases. However, for component refinement where components are fairly coupled, semantic specification is vital.

Unlike the current component technology using UML (Unified Modeling Language) and OCL (Object Constraint Language) [16] to express semantic constraints, constraints in this paper are related to the temporal logic CTL. CTL* logic is formally expressed via two quantifiers **A** ("for all paths") and **E** ("for some path") together with five temporal operators **X** ("next"), **F** ("eventually"), **G** ("always"), **U** ("until") and **R** ("release") [4]. CTL (Computation Tree Logic) is a restricted subset of CTL* in which each temporal operator must be preceded by a quantifier. An incremental verification technique for CTL properties has been attempted by [6, 13]. It is named *open incremental model checking* (OIMC) for the *open* and *incremental* characteristics of the algorithm. Suppose that a base component is refined by another component. The approach consists of the following activities:

1. Deriving a set of *preservation constraints* at the interface states of the base such that if those constraints are preserved, the property inherent to the base under consideration is guaranteed.
2. The refining component does not violate the above property of the base if, during its execution, the above constraints are preserved.

3. DYNAMIC BEHAVIOR SPECIFICATION

There are two types of semantic mentioned in this paper: component dynamic behavior (this section) and component consistency (Section 4).

In the typical case of component refinement, there are two interacting components: *base* and *extension (or refinement)*. Between the base and its extension, on the base side, is an interface consisting of *exit* and *reentry* states [6, 13]. An exit state is the state where control is passed to the extension. A reentry state is the point at which the base regains control. Correspondingly, the extension interface contains *in*- and *out*-states at which the refinement component receives and returns system control. Let AP be a set of atomic propositions. The dynamic behavior of a component is independently represented by a state transition model.

Definition 1. *A state transition model M is represented by a tuple $\langle S, \Sigma, s_0, R, L\rangle$ where S is a set of states, Σ is the set of input events, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \to S$ is the transition function (where $PL(\Sigma)$ denotes the set of guarded events in Σ whose conditions are propositional logic expressions), and $L : S \to 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

A base is expressed by a transition model $B = \langle S_B, \Sigma_B, s_{o_B}, R_B, L_B\rangle$ and an *interface* I. The interface is a tuple of two state sets $I = \langle exit, reentry\rangle$, where $exit, reentry \subseteq S_B$. An extension is similarly represented by a model $E = \langle S_E, \Sigma_E, \perp, R_E, L_E\rangle$. $\perp$ denotes no-care value. Its interface is $J = \langle in, out\rangle$.

E can be semantically plugged with B via *compatible* interface states. Logically, along the computation flow, when the system is in an exit state $ex \in I.exit$ of B matched with an in-state $i \in J.in$ of E, denoted as $ex \leftrightarrow i$, it can enter E if the conditions to accept extension events, namely the set of atomic propositions at i, are satisfied. That is, $\bigwedge L_B(ex) \Rightarrow \bigwedge L_E(i)$, where $\bigwedge$ is the inter-junction of atomic propositions. Similar arguments are made for the matching of a reentry state $re \in I.reentry$ and an out-state $o \in J.out$. The conditions resemble to pre- and post-conditions in *design by contract* [12].

Definition 2. *Within interfaces I and J of B and E, the pairs $\langle ex, i\rangle$ and $\langle re, o\rangle$ can be respectively mapped according to the following conditions.*

- $ex \leftrightarrow i$ *if* $\bigwedge L_B(ex) \Rightarrow \bigwedge L_E(i)$.
- $re \leftrightarrow o$ *if* $\bigwedge L_E(o) \Rightarrow \bigwedge L_B(re)$.

The actual mapping configuration is decided by the modeler at composition time. Subsequently, ex and re will be used in place of i and o respectively in this paper.

Definition 3. *Composing the base B with the extension E, through the interface I produces a composition model $C = \langle S_C, \Sigma_C, s_{0_C}, R_C, L_C\rangle$ as follows:*

- $S_C = S_B \cup S_E$; $\Sigma_C = \Sigma_B \cup \Sigma_E$; $s_{0_C} = s_{0_B}$;
- *R_C is defined from R_B and R_E in which R_E takes precedent, namely any transition in B is overridden by another transition in E if they share the same starting state and input event;*
- $\forall s \in S_B, s \notin I.exit \cup I.reentry : L_C(s) = L_B(s)$;
- $\forall s \in S_E, s \notin J.in \cup J.out : L_C(s) = L_E(s)$;
- $\forall s \in I.exit \cup I.reentry : L_C(s) = L_B(s)$.

In this formal specification, the behavior of B can be partially overridden by E because E takes precedent during composition.

Definition 4. *The* closure *of a property p, $cl(p)$, is the set of all sub-formulae of p including itself.*

- $p \in AP : cl(p) = \{p\}$
- *p is one of* $\mathbf{AX}\, f, \mathbf{EX}\, f, \mathbf{AF}\, f, \mathbf{EF}\, f, \mathbf{AG}\, f, \mathbf{EG}\, f$: $cl(p) = \{p\} \cup cl(f)$
- *p is one of* $\mathbf{A}\,[f\ \mathbf{U}\ g], \mathbf{E}\,[f\ \mathbf{U}\ g], \mathbf{A}\,[f\ \mathbf{R}\ g], \mathbf{E}\,[f\ \mathbf{R}\ g]$: $cl(p) = \{p\} \cup cl(f) \cup cl(g)$
- $p = \neg f : cl(p) = cl(f)$
- $p = f \vee g$ *or* $p = f \wedge g : cl(p) = cl(f) \cup cl(g)$

Definition 5. *The truth values of a state s with respect to a set of CTL properties ps within a model $M = \langle S, \Sigma, s_0, R, L\rangle$, denoted as $\mathcal{V}_M(s, ps)$, is a function: $S \times 2^{CTL} \to 2^{CTL}$.*

- $\mathcal{V}_M(s, \emptyset) = \emptyset$

- $\mathcal{V}_M(s, \{p\} \cup ps) = \mathcal{V}_M(s, \{p\}) \cup \mathcal{V}_M(s, ps)$
- $\mathcal{V}_M(s, \{p\}) = \begin{cases} \{p\} & \text{if } M, s \models p \\ \{\neg p\} & \text{otherwise} \end{cases}$

Hereafter, $\mathcal{V}_M(s, \{p\}) = \{p\}$ (or $\{\neg p\}$) is written in the shorthand form as $\mathcal{V}_M(s, p) = p$ (or $\neg p$) for individual property p.

OIMC is rooted at *assumption model checking* [11]. This method is particularly useful for open systems - future extensions are not known in advance. Hence, OIMC is applicable to component-based software. The idea to the component refinement context is explained in the following. The composite model C can be treated as the combination of two sequential components B and E. In addition to existing execution paths defined in B, a typical execution path in C consists of three parts: initially in B, next in E and then back to B. Associated with each reentry state re of E is a computation tree rooted at the state and lying completely in B. This tree possesses a set of temporal properties. If these properties at re are known, without loss of correctness, we can efficiently derive the properties at the upstream states in E by ignoring model checking in B to find the properties at re. Instead, we start from these reentry states with the associated properties; check the upstream of the extension component, and then the base component if needed [1]. The properties associated with a reentry state re are assumed with truth values from B, $As(re) = \mathcal{V}_B(re, cl(p))$. As is the assumption function of this assumption model checking. Of course, this method relies on whether $As(re)$ is proper.

The assumption As at a reentry state re is *proper* if the seeding values are exactly the properties associated with the tree at re in C, i.e. $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$. The assumption As is definitely proper if re is not affected by E. The problem arises if ex is reachable from re in B. On the other hand, re is reachable from ex in E. This situation creates a circular dependency between interface states ex and re. Dealing with such a circular structure is indeed very important to the verification result of assumption model checking. In fact, this is the weak point of assumption model checking. In this paper, that topic is out of the scope. Subsequent discussions consider As is proper.

4. INTER-COMPONENT CONSISTENCY

Given a structure $B = \langle S_B, \Sigma_B, s_{0_B}, R_B, L_B \rangle$ as in Definition 1, a property p holding in B is denoted by $B, s_{0_B} \models p$. C is formed by composing B and E, $C = B + E$. B and E are *consistent* with respect to p if $C, s_{0_B} \models p$.

4.1 A Theorem on Component Consistency

Due to the inherently inside-out characteristic of model checking, after checking p in B, at each state s, $\mathcal{V}_B(s, cl(p))$ are recorded.

Definition 6. *B and E are in* conformance *at an exit state ex (with respect to $cl(p)$) if $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_E(ex, cl(p))$.*

In this definition, $\mathcal{V}_E(ex, cl(p))$ are derived from the assumption model checking within E, and the seeded values at any reentry state re are $As(re) = \mathcal{V}_B(re, cl(p))$.

[1]There is no need to model check the base again if the consistency constraints associated with the exit states of B are preserved at the respective in-states of E.

Theorem 7. *Given a base B and a property p holding on B, an extension E is attached to B at some interface states. E does not violate property p if B and E conform with each other at all exit states.*

The proof details are in [13]. Even though this paper focuses on component refinement, with regards to COTS, the above theorem also holds. A COTS component can be indeed regarded as a special case of refinement in which there is only a single exit state and no reentry state with the base. The computation tree of the COTS deviates from the base and never joins the base again. After being composed with a COTS, instead of an assumption model checking within the COTS, a standard model checking procedure can be executed entirely within the COTS to find the properties at the exit state. The conformance condition to ensure the consistency between the two components can be applied as usual. The only difference in Definition 6 lies in $\mathcal{V}_E(ex, cl(p))$ for each exit state ex. In component refinement, these truth values are derived from the assumption model checking within E with the assumption values $\mathcal{V}_B(re, cl(p))$ at any reentry state re. On the contrary, in COTS, there is no assumption at all. Hence, the model checking procedure in E is then exactly standard CTL model checking.

Figure 1 depicts the composition preserving the property $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ when B and E are in conformance. The composition is done via a single exit state ex. The reentry state re is not shown but it does not affect the subsequent arguments [2]. E overrides the transition ex-s_3 in B. B' is the remainder of B after removing the overridden transition. In the figure, within B, $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ holds at s_1, s_2 and ex. The figure only shows $\mathcal{V}_E(ex, p) = \mathcal{V}_B(ex, p) = \mathbf{A}\,[f\,\mathbf{U}\,g]$. In fact, B and E conform at ex with regards to $cl(p)$. After removing the edge ex-s_3, the new paths in E together with the remaining computation tree in B' still preserve p at ex directly; and consequently s_2 and s_1 indirectly. As p is preserved at the initial state s_1, B and E are consistent.

By Theorem 7, component semantic specification requires $\mathcal{V}_B(s, cl(p))$ for tuples of any potential interface state s and any CTL property p inherent to B. They serve as constraints for component consistency (Section 5.2).

4.2 Open Incremental Model Checking

Components can be verified to be consistent via OIMC. Initially, a CTL property p is known to hold in B. We need to check that E does not violate p. From Theorem 7, the incremental verification method only needs to verify the conformance at all exit states between B and E. Corresponding to each exit state ex, within E, the algorithm to verify preservation constraints $\mathcal{V}_B(ex, cl(p))$ can be briefly described as follows:

1. Seeding $\mathcal{V}_B(re, cl(p))$ at any reentry state re. The assumption function As is: $As(re) = \mathcal{V}_B(re, cl(p))$.

2. Executing a CTL assumption model checking procedure in E to check ϕ, $\forall \phi \in cl(p)$. In case of COTS, a standard CTL model checking is executed within E instead.

3. Checking if $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$.

[2]In fact, this figure is intended to represent both component refinement and COTS.

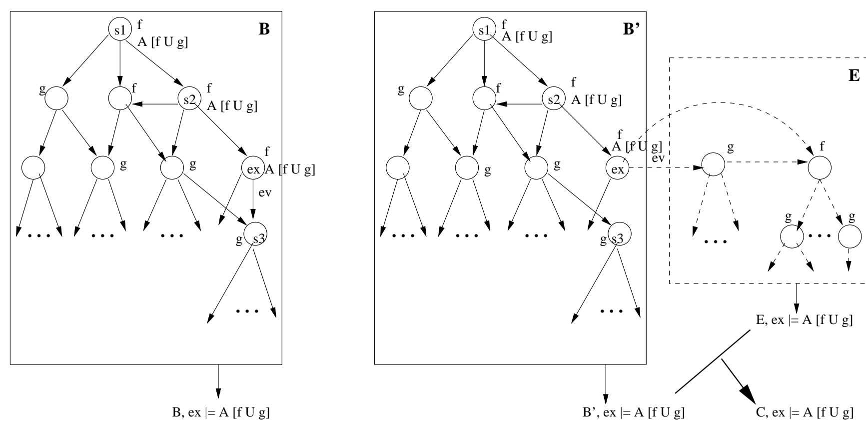


Figure 1: An illustration of conformance $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$ where E overrides B. The property $p = \mathbf{A}\,[f\ \mathbf{U}\ g]$ is preserved in B due to the conformance.

At the end of the algorithm, if at all exit states, the truth values with respect to $cl(p)$ are matched respectively, B and E are consistent with respect to p.

4.3 Scalability of OIMC

This section addresses the scalability of the algorithm in Section 4.2. We consider the general case of the n-th version of the component (C_n) during software evolution as a structure of components B, E_1, E_2, ..., E_n where E_i is the refining component to the $(i-1)$-th evolved version ($C_{(i-1)}$), $i = \overline{1, n}$. The initial version is $C_0 = B$ and $C_i = C_{i-1} + E_i$. We check for any potential conflict between B and E_i regarding p via OIMC. Theorem 8 claims that OIMC is scalable. The detailed proof is in [13].

THEOREM 8. *If all respective pairs of base ($C_{(i-1)}$) and refining (E_i) components conform, the complexity of OIMC to verify the consistency between E_n and B is independent from the n-th version C_n, i.e. it only executes within E_n.*

5. COMPONENT SPECIFICATION

This paper advocates the inclusion of two additional semantic aspects of component specification to facilitate proper component composition. Given a base component $B = \langle S_B, \Sigma_B, s_{o_B}, R_B, L_B \rangle$, the semantic aspects are: dynamic behavior (via state transition model in which only potential future interface states are visible to other components - Section 3) and their associated consistency constraints (via the truth values of $\mathcal{V}_B(s, cl(p))$ at such an interface state s, where p is a CTL property holding in the base component - Section 4).

5.1 Interface Signature

Component signatures are the fundamental aspect to the component interface. As commonly recognized, the traditional interface signature of a component contains *attributes* and *operations*. First, through attributes [3], the current state of a software component may be externally observable. The component's clients can observe and even change the values of those attributes. Second, the environment interacts with the component through operations. The operations represent services or functions the component provides.

Unlike above two static aspects, the introduction of dynamic behavior of a component to the interface is recommended in this paper. Components in reality resemble classes in the object-oriented (OO) approach. This specification style hence follows the encapsulation principle of OO technology so that only essential information is exposed. Only the partial dynamic model of the component consisting of potential future interface states is visible to clients. The rest of the model can be hidden. Associated with a visible interface state s is the set of atomic propositions $L(s)$ (Definition 1). These propositions are often expressed via logic expressions among attributes above.

5.2 Interface Constraints

The interface signature only shows the individual elements of the component for interaction with clients in syntactic terms. In addition to the constraints imposed by their associated types, the attributes and operations of a component interface may be subject to a number of further semantic constraints regarding their use. In general, there are two types of such constraints: internal to individual components and inter-component relationships. The first type is simple and has been thoroughly mentioned in many component-related works [9, 16]. The notable examples are the operation semantics according to pre-/post-conditions of operations; and range constraints on attributes. For the second type, current component technology such as CORBA IDL (Interface Definition Language), UML and OCL [16] etc. is limited to a very weak logic in terms of expressiveness. For example, different attributes in components may be interrelated by their value settings; or an operation of a component can only be invoked when a specific attribute value of another is in a given range etc [9]. The underlying logic only expresses the constraint at the moment an interface element is invoked, i.e. static view, regardless of execution history.

The paper introduces two inter-component semantic constraints. The first constraint is based on the *plugging* compatibility for a refining component to be plugged at a special state of the base. This situation resembles the extension of use-case scenarios. The base gives the basic interacting sce-

[3]Attribute is termed as property in [9] which is essentially the entities expressing states of components. To distinguish them from temporal properties inherent to components in Section 5.2, those entities are named as attributes.

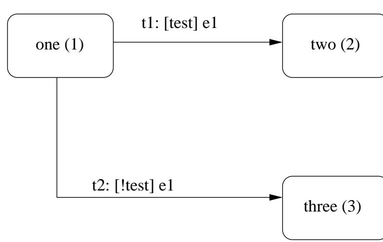


Figure 2: The dynamic behavior model of the "black" component.

narios of the component with clients. The refining component refines some of those scenarios further at a certain point from which the component deviates from the pre-defined course to enter new traces in the extension component. Such a point corresponds to an exit state in Definition 2.

On the other hand, the second semantic constraint emphasizes on how to make components *play* once they are plugged. Importantly, this constraint type is expressed in terms of CTL so its scope of expressiveness is enormous. In contrast to the logic above, CTL can describe whole execution paths of a component, i.e. dynamic view. Via OIMC in Section 4.2, a refining client E to a base component B can be efficiently verified on whether it preserves the property p of B.

Once composed, the new component $C = B + E$ exposes its new interface signatures and constraints. Static aspects like attributes and operations are simply the sum of those in B and E. The dynamic behavior of C is exposed according to the composition of corresponding visible parts of B and E. In terms of constraints, any potential interface state s is exposed with the set of propositions $L_C(s) = L_B(s)$ according to Definition 3. On the other hand, the consistency constraint at s is derived either from $\mathcal{V}_B(s, cl(p))$ (for any $s \in S_B$) or $\mathcal{V}_E(s, cl(p))$ which is resulted from the above execution of OIMC within E (if $s \in S_E$). Subsequent refinements to C follow the same manner as the case of E to B because of Theorem 8.

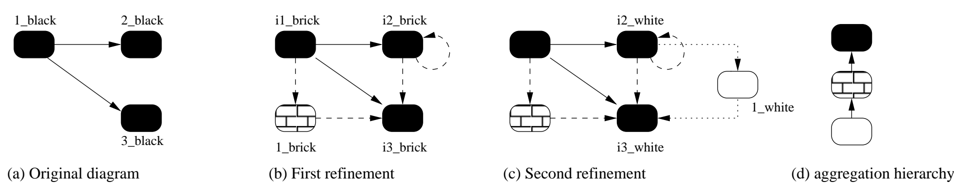
5.3 Component Specification and Composition

Component specification can be represented via interface signatures and constraints written in an illustrative specification language below. The major goal of this language is to minimize the "conceptual distance" between architectural abstractions and their implementation [1]. Encoding state diagrams directly into the interface; and refining existing component specifications in pure programming languages are difficult. Instead, a language similar to that of [1] for declaring and refining state machines in layering manner is used. Based on the exemplary specification, components are implemented as classes in typical object-oriented languages. Component composition is then done via class aggregation/merging. Component attributes and operations are declared in the object-oriented style like C++. The `virtual` keyword is used to only name an element without actual memory allocation. The element will be subsequently mapped to the actual declaration in another component. This mechanism resembles `mergeByName` in Hyper/J [15] in which component entities sharing the same label are merged into a single entity during component composition.

Figure 2 shows the dynamic model of a simple component, while below is the corresponding specification of the component. The interface signatures should declare: edges with name, start state, end state, transition guard and input event; as well as transition action. At the end are the semantic constraints of the component written in both types shown in Section 5.2, namely plugging compatible conditions and inherent temporal properties at potential interface states. For illustration purpose and due to space limitation, this producer-consumer example is very much simplified so that only some key transitions and states are shown. Because of this over-simplified model, the whole dynamic behavior of the component is visible to clients. In practice, regarding the encapsulation principle, only essential part of the model for future extension is visible. The rest of the model is hidden from clients. There are three components: "black" (the base B of Figure 3a with solid transitions - item-producing function); "brick" (the first refinement E of Figure 3b expressed via dashed transitions - variable-size buffer and item-consuming function); and "white" (the second refinement E' of Figure 3c depicted in dotted transitions - optimizing data buffer).

```
Component B {
Signature:
  states 1_black, 2_black, 3_black;

  /* edge declarations */
  edge t1:  1_black -> 2_black
  condition test // OK if adding k items to buffer
  input event e1 // producing k items
  do { produce(k)...  }; /* t1 action */

  edge t2:  1_black -> 3_black;
  ...  /* similarly defined */

  // operations and attributes declaration
  boolean test;
  int cons, prod;// consumed, produced items
  int buffer[];// a bag of data items
  ...
  init(){ state = 1_black; ...};
  produce(n){ prod = prod + n;...};

Constraint:
  /* compatible plugging conditions - CC */
  1_black_cc:  cons = prod;// empty buffer
  2_black_cc:  test = true, cons < prod;
  3_black_cc:  test = false, cons ≤ prod;

  /* Inherent properties - IP */
  1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  3_black_ip:  AG (cons ≤ prod), cons ≤ prod;
}
```

As components are composed with each other, they can be progressively refined/extended in layering manner. The process adds states, actions, edges to an existing component. The original component and each refinement are expressed as separate specifications that are encapsulated in distinct layers. Figure 3 shows this hierarchy: the root component is generated by the specification from Figure 2 or Figure 3a; its immediate refinements are in turn generated from component specifications according to the order in the Figures 3b



(a) Original diagram (b) First refinement (c) Second refinement (d) aggregation hierarchy

Figure 3: Component refinements and component composition via class aggregation.

and 3c.

```
Component E {/* for refining black */
Signature:
  states 1_brick, i1_brick, i2_brick, i3_brick;

  /* edges declaration */
  edge t3:  i2_brick -> i3_brick
  condition ...  // ready to consume
  input event ...  // consuming k items
  do { consume(k)...  }; /* t3 action */

  edge t4:  i1_brick -> 1_brick
  condition ...  // ready to change buffer size
  input event ...  // change the size
  do { changesize();...  }; /* t4 action */

  edge t5:  1_brick -> i3_brick;
  edge t6:  i2_brick -> i2_brick;
    // buffer inquiry only, consuming zero item
  ...  /* similarly defined */

  // operations and attributes declaration
  virtual int cons;// mapped with cons in B
  virtual int prod;// mapped with prod in B
  virtual int buffer[];// mapped with buffer in B
  consume(n){ cons = cons + n;...};
  changesize(){ buffer = malloc();...};

Constraint:
  1_brick_cc:  cons ≤ prod;
  i1_brick_cc:  cons ≤ prod;
  i2_brick_cc:  test = true, cons < prod;
  i3_brick_cc:  test = false, cons < prod;
}

Component E' {/* for refining black + brick */
Signature:
  states 1_white, i2_white, i3_white;

  /* edges declaration */
  edge t7:  i2_white -> 1_white
  condition ...  // ready to compact buffer
  input event ...// compact the data buffer
  do { resetbuffer();...  }; /* t7 action */

  edge t8:  1_white -> i3_white;
  ...  /* similarly defined */

  // operations and attributes declaration
  virtual int cons;// mapped with cons in B
  virtual int prod;// mapped with prod in B
  virtual int buffer[];// mapped with buffer in B
  resetbuffer(){ prod = prod - cons; cons = 0;...};

Constraint:
  1_white_cc:  cons ≤ prod, cons = 0;
  i2_white_cc:  test = true, cons ≤ prod;
  i3_white_cc:  test = false, cons ≤ prod;
}
```

Aggregation then plays a central role in this component implementation style. All the states and edges in Figure 3a are aggregated with the refinement of Figure 3b; and this figure is in turn united with the refinement of Figure 3c. The component to be executed is created by instantiating the bottom-most class of the refinement chain of Figure 3d.

The following explains the preservation of the constraint in B by all subsequent two component refinements E and E'. Informally, the property means that under any circumstance, the number of produced items by the component is always greater or equal to that of consumed items. In terms of CTL notation, $p = \mathbf{AG}\,(cons \leq prod)$. The closure set of p is hence $cl(p) = \{p, a\}$, where $a = (cons \leq prod)$.

Initially, B is composed with E. Interface plugging conditions are used to map compatible interface states among components. The base exposes three interface states 1_black, 2_black and 3_black. On the other hand, the refinement component exposes four interface states, namely 1_brick, $i1_brick$, $i2_brick$ and $i3_brick$. Based on the respective atomic proposition sets at those states, corresponding interface states are mapped accordingly.

For instance, first $\bigwedge L_B(1_black) = (cons = prod) \Rightarrow \bigwedge L_E(i1_brick) = (cons \leq prod)$. According to Definition 2, $i1_brick \leftrightarrow 1_black$. On the other hand, because $\bigwedge L_E(i2_brick) = \bigwedge L_B(2_black)$, $i2_brick \leftrightarrow 2_black$. Similarly, $\bigwedge L_E(i3_brick) \Rightarrow \bigwedge L_B(3_black)$, $i3_brick \leftrightarrow 3_black$. Here, $i1_brick$ and $i2_brick$ perform exit states of the base component, while $i2_brick$ and $i3_brick$ are reentry states.

The composite model of the two components $C_1 = B + E$ is shown in Figure 3b. After the designer decides on the mapping configuration between interface states, and properly resolves any mismatches at the syntactic level between B and E, the semantic constraint of consistency between the two due to p is in focus. The OIMC algorithm in Section 4.2 is applied as follows:

1. Copying $\mathcal{V}_B(s, cl(p))$ to the respectively mapped out-states $i2_brick$ and $i3_brick$ in E for any reentry state s such as 2_black and 3_black.

2. Executing assumption model checking within E to find $\mathcal{V}_E(i1_brick, cl(p))$ and $\mathcal{V}_E(i2_brick, cl(p))$. Note that, the model checking procedure is executed within the dashed part in Figure 3b. The solid transitions belong to the base component B and are hence ignored.

3. Checking if $\mathcal{V}_E(i1_brick, cl(p)) = \mathcal{V}_B(1_black, cl(p))$ and

$\mathcal{V}_E(i2_brick, cl(p)) = \mathcal{V}_B(2_black, cl(p))$. If so, B and E conform.

The model checking is very simple and hence its details are skipped. At the end, B and E components conform at all exit states. According to Theorem 7, p is preserved by the second component after evolving to $C_1 = B + E$.

C_1 is then extended with E'. Notably, the interface of the new component C_1 is derived from B and E as below:

```
Component C1 {
Signature:
  states 1_black, 2_black, 3_black, 1_brick;

  /* edge declarations */
  edge t1:  1_black -> 2_black;
  edge t2:  1_black -> 3_black;
  edge t3:  2_black -> 3_black;
  edge t4:  1_black -> 1_brick;
  edge t5:  1_brick -> 3_black;
  edge t6:  2_black -> 2_black;
  /* identical to each component's declaration */

  // operations and attributes declaration
  boolean test;
  int cons, prod;// consumed, produced items
  int buffer[];
  init(){ state = 1_black; ...}
  consume(n){ cons = cons + n;...};
  produce(n){ prod = prod + n;...};
  changesize(){ buffer = malloc();...};

Constraint:
  /* compatible plugging conditions - CC */
  1_black_cc:  cons = prod;
  2_black_cc:  test = true, cons < prod;
  3_black_cc:  test = false, cons ≤ prod;
  1_brick_cc:  cons ≤ prod;

  /* Inherent properties - IP */
  1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  3_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  1_brick_ip:  AG (cons ≤ prod), cons ≤ prod;
}
```

The approach in composing E' with C_1 is similar to the above, we have the following mapping configuration between interface states: $i2_white \leftrightarrow 2_black$, $i3_white \leftrightarrow 3_black$. The same result is achieved, p is preserved by E'. More importantly, the verification method is executed within E' only, i.e. the dotted part in Figure 3c. After composing E', the component becomes $C_2 = C_1 + E'$ shown below:

```
Component C2 {
Signature:
  states 1_black, 2_black, 3_black, 1_brick, 1_white;

  /* edge declarations */
  edge t1:  1_black -> 2_black;
  edge t2:  1_black -> 3_black;
  edge t3:  2_black -> 3_black;
  edge t4:  1_black -> 1_brick;
  edge t5:  1_brick -> 3_black;
  edge t6:  2_black -> 2_black;
  edge t7:  2_black -> 1_white;
  edge t8:  1_white -> 3_black;
  /* identical to each component's declaration */

  // operations and attributes declaration
  boolean test;
  int cons, prod;// consumed, produced items
  int buffer[];
  init(){ state = 1_black; ...}
  consume(n){ cons = cons + n;...};
  produce(n){ prod = prod + n;...};
  changesize(){ buffer = malloc();...  };
  resetbuffer(){ prod = prod - cons; cons = 0;...};

Constraint:
  /* compatible plugging conditions - CC */
  1_black_cc:  cons = prod;
  2_black_cc:  test = true, cons < prod;
  3_black_cc:  test = false, cons ≤ prod;
  1_brick_cc:  cons ≤ prod;
  1_white_cc:  cons ≤ prod, cons = 0;

  /* Inherent properties - IP */
  1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
  3_black_ip:  AG (cons ≤ prod); cons ≤ prod;
  1_brick_ip:  AG (cons ≤ prod); cons ≤ prod;
  1_white_ip:  AG (cons ≤ prod); cons ≤ prod;
}
```

In brief, p is preserved by both extensions E and E'. In this example, the scalability of incremental model checking is maintained as it only runs on the refinements, independently from the bases B and C_1 respectively.

6. RELATED WORK

Modular model checking is rooted at assume-guarantee model checking [10, 14]. However, unlike the counterpart in hardware verification [8, 10] focusing on parallel composition of modules, software modular verification [11] is restricted by its sequential execution nature. Incremental model checking inspires verification techniques further. There is a fundamental difference between those conventional modular verification works [8, 10, 14] and the proposed approach including this paper and [6]. Modular verification in the former works is rather closed. Even though it is based on component-based modular model checking, it is not prepared for change. If a component is added to the system, the whole system of many existing components and the new component are re-checked altogether. On the contrary, the OIMC approach in this paper and [6] is incrementally modular and hence more open. It only checks the new system's consistency within the new component. Certainly, this merit comes at the cost of "fixed" preservation constraints at exit states. These constraints can deliver a false negative for some cases of component conformance.

Regarding the assumption aspect in component verification, [7] presents a framework for generating assumption on environments in which the component satisfies its required property. This work differs OIMC in some key points. First, the constraints in OIMC are explicitly fixed at $\mathcal{V}_B(ex, cl(p))$ for any exit state ex, whereas based on a fully specified component model including error states, [7] generates assumption about operation calls by which the environment

does not lead the component to any error state. Second, the approach in [7] is viewed from a static perspective, i.e. the component and the external environment do not evolve. If the component changes after adapting some refinements, the assumption-generating approach is re-run on the whole component, i.e. the component model has to be re-constructed; and the assumption about the environment is then generated from that model.

The OIMC technique introduced in this paper is similar to [6]. However, our work proposes more precisely and explicitly the conformance condition between components. Further, to enable the plug-and-play idea in component-based software, the corresponding component specification are separately specified. Their composition is based on plugging condition among compatible interface states. In addition, the scalability of component consistency is not mentioned in [6]. Without Theorem 8, the approach is not applicable for future component composition.

Finally, like the proposal in Section 5 about encapsulating dynamic behavior model into component interface, i.e. state-full interface, two closely related works [3, 5] also advocate the use of *light-weight formalism* to capture temporal aspects of software component interfaces. More specifically, this paper simply relies on state transition model in the most general sense, while the approach in [3, 5] presents a finer realization of state-full model in which states are represented by control points in operations of components; and edges are actually operation calls. That approach focuses on the order of operation calls in a component [4]. By formalizing a component through a set of input, output and internal operations, the compatibility between component interfaces with regards to the structure of component operations is defined and checked. In addition, the two approaches target different aspects of consistency. This paper is concerned with component consistency in terms of CTL properties, whereas the approach in [3, 5] is involved with the correctness and completeness of operation declarations within components.

7. CONCLUSION

This paper focuses on the refinement aspect of components in which components are relatively coupled. However, the results of this paper can be equally applied to COTS. This paper advocates the inclusion of *dynamic behavior* and *component consistency* written in CTL to the component interface to better deal with component matching. Besides the traditional static elements such as operations and attributes, component interface should include potential interface states together with the associated plugging conditions and consistency constraints at those states. Next, based on the proposed specification structure, an efficient and scalable model checking method (OIMC) is utilized to verify whether components are consistent.

Current well-known model checkers do not support assumption model checking. A future work is to encapsulate the assumption feature into an open-source model checker such as NuSMV [2].

8. REFERENCES

[1] D. Batory, C. Johnson, B. MacDonald, and D. V. Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proc. International Conference on Software Reuse*, July 2000.

[2] R. Cavada, A. Cimatti, G. Keighren, et al. *NuSMV 2.2 Tutorial*. CMU and ITC-irst, nusmv@irst.itc.it, 2004.

[3] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F. Y. C. Mang. Interface compatibility checking for software modules. In *Proceedings of the Computer-Aided Verification - CAV*. LNCS Springer-Verlag, 2002.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[5] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[6] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. Symposium on the Foundations of Software Engineering*, September 2001.

[7] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the International Conference on Automated Software Engineering*, 2002.

[8] O. Grumberg and D. E. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes of Computer Science*. Springer-Verlag, 1991.

[9] J. Han. An approach to software component specification. In *Proceedings of International Workshop on Component Based Software Engineering*, 1999.

[10] O. Kupferman and M. Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[11] K. Laster and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Constructions and Analysis of Systems*, 1998.

[12] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.

[13] T. T. Nguyen and T. Katayama. Handling consistency of software evolution in an efficient way. In *Proc. IWPSE*, pages 121–130, 2004.

[14] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes of Computer Science*. Springer-Verlag, 1999.

[15] P. Tarr and H. Ossher. *Hyper/J(TM) User and Installation Manual*. IBM Research, IBM Corp., 2000.

[16] J. Warmer and A. Kleppe. *The Objects Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[4] In [3], operations are named as methods.