Scenario-based Verification
for Component-based Embedded Software Designs

Jun Hu, Xiaofeng Yu, Yan Zhang, Tian Zhang, Wang Linzhang,
Xuandong Li, Guoliang Zheng

Postprint Version. Originally Published in: International Conference on Parallel Processing 2005,
Pages 240 - 247

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.
Scenario-based Verification for Component-based Embedded Software Designs

Hu Jun, Yu Xiaofeng, Zhang Yan, Zhang Tian, Wang Linzhang, Li Xuandong, and Zheng Guoliang
State Key Laboratory of Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, Jiangsu, P.R.China 210093
hujun@seg.nju.edu.cn

Abstract
In this paper, for embedded software systems we consider the problem of checking component-based designs for scenario-based specifications. The component-based designs are modelled by interface automaton networks which consist of a set of interface automata synchronized by shared actions, and the scenario-based specifications are specified by UML sequence diagrams. Based on investigating the reachability graph of the state space of the interface automaton networks, we develop the algorithms to check the existential consistency and the mandatory existential consistency including the forward, backward, and bidirectional mandatory existential consistency.

1. Introduction

Embedded systems are used in various application areas, from telecommunications to automotive, medical applications, manufacturing, transportation etc. These systems are often required a higher reliability than those general purpose computing systems. And the complexity of embedded system designs is increasing rapidly in recent years. A prevalent trend in the embedded computing area is that systems are more and more dominated by software. The software determines the functionality and quality of systems. As a result of the growing complexity and reliability requirements, the embedded software systems present many challenges to current computing technology.

For embedded software, component-based design has become an important approach for handling complexity, increasing productivity and product quality. Typically, it involves a set of reusable components which interact with each other to produce the expected global behaviors. For

such systems, the reliability mostly depends on the data type compatibility and behavior compatibility among component interfaces. One of major goals in developing such systems is the validation that whether the component-based designs fulfill certain specifications. Interface automaton [4] is a light-weight formal language which can be used in system design phase to model the dynamic temporal behavior of software component interface. Since the interfaces are often much simpler than the corresponding implementations, one of the advantages of interface automata is that the state space of the model will be much less than the corresponding statecharts of implementations.

Using models in embedded software development process is essential for guaranteeing quality and reliability in a cost-effective manner. Model Driven Architecture (MDA)[7] has been established as an important paradigm in the development of software systems today. The main principle of the paradigm is to use models all along the development cycle, from design to implementation. In MDA, Unified Modeling Language (UML)[1] is used as a general purpose visual modeling language. One of the UML kernel models is the sequence diagram, which describes the collaboration of interacting components. Through describing the exchange of a set of messages, it can represent a scenario-based specification which offers an intuitive and visual way for expressing design requirements.

In this paper, for embedded software systems we consider the problem of checking the component-based designs for scenario-based specifications. The design models are represented by interface automaton networks which consist of a set of interface automaton synchronized by shared actions, and the scenario-based specifications are specified by UML sequence diagrams. Those specifications describe the general existential consistency and mandatory existential consistency including the forward, backward, and bidirectional mandatory existential consistency.

The paper is organized as follows. In Section 2 and 3, we introduce UML sequence diagrams and interface automaton networks, and give their formal definitions for verification.
In Section 4, based on investigating the reachability graph of the state space of the design models, we develop the algorithms to check interface automaton networks for those interesting properties. The last section discusses the related work and gives some conclusions.

2. UML sequence diagrams

In component-based system design, an UML sequence diagram can be used to describe an interaction, which is a set of messages exchanged among components within a collaboration to gain a desired operation or result. Its focus is on the temporal order of the message flow. Here we just consider simple sequence diagrams, which describe scenarios without any alternative and loop. For example, a simple sequence diagram is depicted in Fig.1.

![Figure 1. A simple UML sequence diagram](image)

By events we mean message sending, message receiving, creating an component instance or deleting an component instance. Without loss of generality, according to [11] we assume that each sequence diagram corresponds to a visual instance. Without loss of generality, according to [11] we assume that each sequence diagram corresponds to a visual order displayed in

\[
C, E, M, L, W
\]

where \(C\) is a finite set of components; \(E\) is a finite set of events; \(M\) is a finite set of messages. For any message \(g \in M\), let \(g!\) and \(g?\) represent the sending and receiving for \(g\) respectively. For any \(e \in E\), it is corresponding to a send or receive for a message \(g\), denoted by \(\phi(e) = g!\) or \(\phi(e) = g?\).

\(L: E \rightarrow C\) is labelling function which maps each event \(e \in E\) to a component \(L(e) \in C\);

\(W\) is a finite set whose elements are of the form \((e, e')\) where \(e\) and \(e'\) are in \(E\) and \(e \neq e'\) which represents a visual order displayed in \(D\).

The temporal order of the message flow is described by the event sequence of sequence diagrams, which is called the trace of sequence diagrams. Any event sequence is the form of \(e_0, e_1, \ldots, e_m\), which represents that \(e_{i+1}\) takes place after \(e_i\) for any \(i\) (\(0 \leq i \leq m - 1\)).

**Definition 1** A sequence diagram is a tuple \(D = (C, E, M, L, W)\) where

- \(C\) is a finite set of components;
- \(E\) is a finite set of events;
- \(M\) is a finite set of messages. For any message \(g \in M\), let \(g!\) and \(g?\) represent the sending and receiving for \(g\) respectively. For any \(e \in E\), it is corresponding to a send or receive for a message \(g\), denoted by \(\phi(e) = g!\) or \(\phi(e) = g?\);
- \(L: E \rightarrow C\) is labelling function which maps each event \(e \in E\) to a component \(L(e) \in C\);
- \(W\) is a finite set whose elements are of the form \((e, e')\) where \(e\) and \(e'\) are in \(E\) and \(e \neq e'\), which represents a visual order displayed in \(D\).

The temporal order of the message flow is described by the event sequence of sequence diagrams, which is called the trace of sequence diagrams. Any event sequence is the form of \(e_0, e_1, \ldots, e_m\), which represents that \(e_{i+1}\) takes place after \(e_i\) for any \(i\) (\(0 \leq i \leq m - 1\)).

**Definition 2** For any sequence diagram \(D = (C, E, M, L, W)\), a event sequence \(e_0, e_1, \ldots, e_m\) is a trace of \(D\) if the following conditions hold:

- all events in \(E\) occur in the sequence, and each event occurs only once, i.e. \(\{e_0, e_1, \ldots, e_m\} = E\) and \(e_i \neq e_j\) for any \(i, j (i \neq j, 0 \leq i, j \leq m)\);
- \(e_0, e_1, \ldots, e_m\) satisfy the visual order defined by \(W\), i.e. for any \(e_i\) and \(e_j\), if \((e_i, e_j) \in W\), then \(0 \leq i < j \leq m\).

3. Interface automata and component-based designs

3.1. Interface automata

Interface automaton is a light-weight formal language to describe the temporal aspects of software component interfaces. The basic idea is an optimistic view on composition of components. That is, when the components are composed, their environment assumptions should be also composed. Accordingly, if there are some helpful environments satisfy both of their environment assumptions, these components are considered compatible. Specifically, interface automata are designed to capture effectively both input assumptions and output guarantees about the order of the interactions between component and its environment. For example, Fig.2 shows an interface automaton of a communication component [4]. In state 0, it only accepts the input msg, indicating the assumption that once the method msg is called, the environment will wait for an ok or fail response before another call of msg. The formal definitions are given below.
Figure 2. Interface automata of a communication component

Definition 3 An interface automaton is a tuple \( P = (V_P, v_{P}^{init}, A_P, A_O^P, A_H^P, \Gamma_P) \) where:
- \( V_P \) is a finite set of states, each state \( v \in V_P \);
- \( v_{P}^{init} \in V_P \) is the initial state;
- \( A_P^I, A_O^P \) and \( A_H^P \) are mutually disjoint sets of input, output and internal actions, and \( A_P = A_P^{I} \cup A_P^{O} \cup A_P^{H} \) is the set of all actions;
- \( \Gamma_P \subseteq V_P \times A_P \times V_P \) is a set of transitions.

\[ \text{If } a \in A_P^I, \text{ resp. } a \in A_P^O, \text{ then } (v_i, a, v_j) \text{ is called an input (resp. output, internal) transition.} \]

Definition 4 For an interface automaton \( P = (V_P, v_{P}^{init}, A_P, A_O^P, A_H^P, \Gamma_P) \), a state sequence \( v_0 \xrightarrow{a_0} v_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} v_n \xrightarrow{a_n} v_{n+1} \) is a behavior of \( P \) iff \( v_0 = v_{P}^{init} \), and for each \( i(0 \leq i \leq n) \), there is \( (v_i, a_i, v_{i+1}) \in \Gamma_P \).

3.2. Interface automaton networks

We use interface automaton networks to model the designs of component-based embedded software system. A IANs consists of a set of interface automata which represent the abstractions of software components. An input action of one interface automaton may coincide with an output action of the other one, then these two interface automata will synchronize on such shared actions, asynchronously interleaving on other actions. Those synchronized actions between any two interface automata are denoted by \( shared(P_i, P_j) = A_{P_i} \cap A_{P_j} = (A_{P_i}^{O} \cup A_{P_j}^{H}) \cup (A_{P_j}^{O} \cup A_{P_i}^{H}) \). The states, actions and transitions of the IANs are defined as follows. More details about composition of interface automata can be found in [4].

Definition 5 Interface automaton networks (IANs) is a tuple \( N = (K, S) \), where
- \( K = \{P_1, P_2, \ldots, P_n\} \) is a set of composable interface automata;
- \( S = \{shared(P_i, P_j) \mid 1 \leq i, j \leq n, i \neq j\} \) is a set of all shared actions.

Definition 6 Let \( N = (K, S) \) be an IANs where \( K = \{P_1, P_2, \ldots, P_n\} \),
- a state \( \tau \) of \( N \) is in \( V_{P_1} \times V_{P_2} \times \cdots \times V_{P_n} \), that is, \( \tau = (v_1, v_2, \ldots, v_n) \) where \( v_i \in V_{P_i}, 1 \leq i \leq n \);
- the initial state of \( N \) is \( v_{P_1}^{init} \times V_{P_2} \times \cdots \times v_{P_n}^{init} \), and \( v_{N}^{init} = (v_{P_1}^{init}, v_{P_2}^{init}, \ldots, v_{P_n}^{init}) \);
- the set of actions of \( N \) is \( A_N = A_N^{I} \cup A_N^{O} \cup A_N^{H} \), where the set of input actions is \( A_N^{I} = (\bigcup_{1 \leq i \leq n} A_{P_i}^{I}) / S \), the set of output actions is \( A_N^{O} = (\bigcup_{1 \leq i \leq n} A_{P_i}^{O}) / S \), and the set of internal actions is \( A_N^{H} = (\bigcup_{1 \leq i \leq n} A_{P_i}^{H}) / S \).

Definition 7 Let \( \tau = (K, S) \) be an IANs, \( \tau \) and \( \tau' \) be its states where \( \tau = (v_1, v_2, \ldots, v_n) \) and \( \tau' = (v'_1, v'_2, \ldots, v'_n) \). We say that from the state \( \tau \) the system can change to the state \( \tau' \) by an transition \( \tau \xrightarrow{a} \tau' \) if one of the following conditions holds:
- for an action \( a \not\in shared(P_i, P_j) \) for \( 1 \leq i, j \leq n, i \neq j \), there is a transition \( (v_k, a, v'_k) \in \Gamma_{P_k} \) in \( P_k \) for \( 1 \leq k \leq n \), and \( v_i = v'_i \) for any \( i(k \neq i, 1 \leq k \leq n) \);
- for an action \( a \in shared(P_i, P_j) \) for \( 1 \leq i, j \leq n, i \neq j \), there is a transition \( (v_i, a, v'_i) \in \Gamma_{P_i} \) for \( i(k \neq i, 1 \leq k \leq n) \), and \( v_j = v'_j \) for any \( j(k \neq j, 1 \leq k \leq n) \).

Notice that one automaton may produce an output event that is an input event of another automaton, but it is not accepted by the latter one. In fact, this case shows that there are some contradictions between the environment assumptions of these two automata which have shared actions. When this happens, those corresponding states of \( N \) are called illegal states. An algorithm has been presented to compute the compatible composition of interface automata by removing the set of illegal \( N \) recursively[4]. As a result we can denote by \( comp(N) \) the compatible IANs.

Definition 8 Let \( comp(N) = (K, S) \) be a compatible IANs. A state sequence \( \tau_0 \xrightarrow{a_0} \tau_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} \tau_{n+1} \) is a behavior of \( comp(N) \) iff \( \tau_0 = \tau_{\text{init}} \), and for each \( i(0 \leq i \leq n) \), there is \( (\tau_i, a_i, \tau_{i+1}) \in \Gamma_{\text{comp}(N)} \).

4. Checking component-based designs for scenario-based specifications

Now the component-based designs are modelled by the IANs, and the scenario-based specifications are specified
by the sequence diagrams which describe the interactions among components. In the following, we mainly concern about the problem of checking existential consistency between the specification and design models. Specifically, four kinds of existential consistency are considered:

- **General existential consistency**: if a scenario described by a given sequence diagram \( D \) occurs at least once in the behaviors of \( \text{comp}(N) \), or if any forbidden scenario described by a given sequence diagram \( D \) never happens in the behaviors of \( \text{comp}(N) \);

- **Forward mandatory existential consistency**: if a conditional scenario described by a given sequence diagram \( D_1 \) occurs, then a scenario described by the other given sequence diagram \( D_2 \) must follow that occurrence successively in a behavior of \( \text{comp}(N) \);

- **Backward mandatory existential consistency**: if a certain scenario described by a given sequence diagram \( D_1 \) occurs, then this occurrence must follow successively from a scenario described by another given sequence diagram \( D_2 \) also follows, then the middle subsequence between these two scenarios must exactly correspond to a scenario described by the third given sequence diagram \( D_3 \) in a behavior of \( \text{comp}(N) \).

Essentially we need to compare the event sequences of the behaviors of \( \text{comp}(N) \) with the traces of corresponding sequence diagrams. However, the behaviors in the state space of design models may be infinite, the key point is to find a way to check those existential consistency properties in a finite manner.

### 4.1. Projections and simple paths

Let \( \text{comp}(N) = (K, S) \) is a compatible IANs, and \( \varrho = \tau_0 \xrightarrow{a_0} \tau_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} \tau_{n+1} \) is a behavior of \( \text{comp}(N) \). For the action sequence \( a_0, a_1, \ldots, a_{n-1}, a_n \), each internal action \( a_i \) (\( 0 \leq i \leq n \)) which is also a shared action \( a_i \in S \) can be replaced by a pair of input and output actions \( a_i^? a_i! \) (\( 0 \leq i \leq n \)), then we get an action sequence which only contains input and output actions. In fact, for those consistency checking problems, each input/output action \( a \) can be treated as an input/output event \( e \), thus a corresponding event sequence \( e_0 e_1^* \ldots e_r \) \( (r \geq n) \) is obtained, which is called the trail of the behavior \( \varrho \). Notice that any message in the sequence diagrams actually should be corresponding to one of the shared actions in \( \text{comp}(N) \), and each event in the sequence diagrams such as sending message or receiving message should be one of the output or input actions of the corresponding interface automaton. However, in the formal definition of the sequence diagram \( D = (C, E', M, L, W) \), an event \( e' \in E' \) just represents an event name and the event content is denoted by \( \phi(e') \). Thus if we want to compare the events between the trail of \( \varrho \) and the trace of \( D \), we need to compare \( \varrho \) with \( \phi(e') \).

Let \( \sigma \) be the trail of a behavior \( \varrho \), and \( \sigma_1 \) be a subsequence of \( \sigma \) of the form \( e_0 e_1^* \ldots e_m \). For a trace of \( D \) of the form \( f_0^* f_1^* \ldots f_n \), if there are \( e_{k_0}, e_{k_1}, \ldots, e_{k_n} \) satisfying that

- \( 0 = k_0 < k_1 < \ldots < k_n = m \);
- \( \phi(f_0) = e_{k_0} \); and for any other \( \phi(f_i) (0 < i \leq n) \), there is only one \( e_{k_j} = \phi(f_i) (0 < j \leq n) \); and
- for any \( ii(0 \leq i \leq n) \), for any \( j \neq k_p (0 \leq j \leq m, 0 \leq p \leq n) \), \( e_j = \phi(f_i) \),

then we say that \( \sigma_1 \) is a projection of \( \sigma \) over \( D \). If \( \sigma_1 \) further satisfies that \( \phi(f_i) = e_{k_i} \) for any \( i(0 \leq i \leq n) \), we say that \( \sigma_1 \) is a legal projection of \( \sigma \) over \( D \). In this case, the subsequence \( \sigma_1 \) actually represents an occurrence of the scenario described by \( D_1 \).

Based on the \( \text{comp}(N) \), a corresponding reachability graph \( G = (V, T) \) can be constructed easily as follows, where \( V \) is a set of nodes and \( T \) is a set of edges:

- for each compatible state \( \tau_i \) of \( \text{comp}(N) \), there is a node \( v_i \) in \( V \); and for the initial state \( \tau_0^{\text{init}} \) of \( \text{comp}(N) \), the corresponding \( v_0 \) in the set \( V \) is called initial node;

- for each transition \( (\tau_i, a_i, \tau_{i+1}) \in \Gamma_{\text{comp}(N)} \), there is an edge \( l_i = (v_i, a_i, v_{i+1}) \) in \( T \), where \( l_i \) is the label on the edge, and \( l_i = a_i \).

Obviously for any behavior of \( \text{comp}(N) \), there is a path in the reachability graph \( G \). Let \( \rho = t_0^* t_1^* \ldots t_n^* \) be a path of \( G \), and its label sequence is \( l_0^* l_1^* \ldots l_n^* \). Any edge sequence \( t_i^* t_{i+1}^* \ldots t_{i+k}^* (0 \leq i \leq n-k) \) is called a subpath of \( \rho \).

For the label sequence of the form \( l_0^* l_1^* \ldots l_m^* \) of the path \( \rho \), correspondingly there is an action sequence \( a_0^* a_1^* \ldots a_m^* \). As being mentioned before, for those \( a_i \), satisfying that \( a_i \) is a shared action, replacing them with a pair of input and output actions \( (a_i^?, a_i!) \), then again we get an action sequence \( a_0^* a_1^* \ldots a_m^* \) \( (s \geq m) \) and the corresponding event sequence \( e_0 e_1^* \ldots e_r (r \geq n) \), which is called the trail of path \( \rho \). We denote the trail of subpath \( t_i^* t_{i+1}^* \ldots t_j^* \) by \( \sigma(l_i, t_j) (i \leq j) \). Thus each label either is an input(output) event or is a pair of input and output events. We denote by \( \psi(l_k) \) the event pair corresponding to \( l_k \).

It is clear that for any event sequence \( \sigma \) which is the trail of a behavior of \( \text{comp}(N) \), there is a path \( \rho \) in \( G \).
whose trail is \( \sigma \) exactly. So basically the problem of checking \( \text{comp}(N) \) for a sequence diagram \( D \) can be solved by checking the paths in the reachability graph \( G \) for \( D \). However, the length of a path in \( G \) could be infinite and the number of paths could also be infinite, we have to manage the problem based on a finite set of finite paths in \( G \).

For a sequence diagram \( D = (C, E, M, L, W) \), a path \( \rho = t_0t_1\ldots t_n \) with a tail \( \sigma(t_0, t_n) \) is called a simple path over \( D \) if it satisfies the following conditions:

- the trail \( \sigma(t_i, t_n)(0 \leq i \leq n) \) is a projection of \( \sigma(t_0, t_n) \) over \( D \);
- for any \( j \), \( 0 \leq j \leq n, \rho_j \neq t_k \); and
- for any neighbor \( j, k(i \leq j < k \leq n, \psi(t_j) \in E, \psi(t_k) \in E), \rho_j \neq t_k(j < g < h < k) \).

The first condition shows that the end part of \( \rho \) may include exactly an occurrence of a scenario described by \( D \). The second and the third conditions constrain on loops in different segments of \( \rho \). Obviously the length of a simple path in \( G \) is finite and the number of simple paths in \( G \) is also finite. And we say that a simple path in reachability graph \( G \) satisfies a sequence diagram \( D \) if and only if the projection \( \sigma(t_i, t_n) \) is a legal projection.

For a sequence diagram \( D \), a path \( \rho \) in \( G \) is called a prefix for a simple path if it may be extended into a simple path over \( D \), i.e. there could be a subpath \( \rho' \) such that \( \rho \rho' \) is a simple path. By checking whether the path that we have so far traversed is a prefix for a simple path or not, all simple paths in \( G \) can be found out.

### 4.2. Checking general existential consistency

For the general existential consistency checking problem, we consider that if a scenario described by a given sequence diagram \( D \) occurs at least once in the behaviors of the \( \text{comp}(N) \), or if any forbidden scenario described by a given sequence diagram \( D \) never happens in the \( \text{comp}(N) \), which is depicted in Fig.3. Based on the discussion in the above subsection, the following theorem can be established.

\[
\begin{align*}
\text{Theorem 1} & \quad \text{Let } G \text{ be a reachability graph of } \text{comp}(N). \text{ For a sequence diagram } D, \text{ comp}(N) \text{ satisfies the general existential consistency iff there is a simple path of } G \text{ which satisfies } D. \\
\end{align*}
\]

According to the theorem 1, an algorithm is developed to check the general existential consistency of \( \text{comp}(N) \) for a sequence diagram \( D \), which is showed in Fig.4.

\[
\begin{align*}
& \text{current\_path} := (v_0); \\
& \text{satisfied} := \text{false}; \\
& \text{error\_trace} := \emptyset; \\
\text{repeat} & \quad \begin{aligned}
& \text{node} := \text{the last node of current\_path}; \\
& \text{if node has no new successive node} \\
& \quad \text{then delete the last node of current\_path}; \\
& \text{else} \\
& \quad \text{begin} \\
& \quad \text{node} := \text{a new successive node of node}; \\
& \quad \text{if node is such that the path corresponding to current\_path becomes simple path over } D \\
& \quad \quad \text{then} \\
& \quad \quad \text{begin} \\
& \quad \quad \text{if the simple path satisfies } D \\
& \quad \quad \quad \text{then satisfied} := \text{true}; \\
& \quad \quad \text{else put the subpath whose trail is not a legal projection into error\_trace}; \\
& \quad \quad \text{end} \\
& \quad \quad \text{if node is such that the path corresponding to current\_path is a prefix for a simple path over } D \\
& \quad \quad \text{then append node to current\_path}; \\
& \quad \quad \text{end} \\
& \quad \quad \text{until current\_path} = \emptyset; \\
& \quad \text{if satisfied} \text{ then return true else return false.} \\
\end{aligned}
\end{align*}
\]

\[
\text{Figure 4. Algorithm for checking general existential consistency}
\]

Now the details of the algorithm is given below. the algorithm traverses the reachability graph \( G \) of \( \text{comp}(N) \) in a depth first manner, starting from the initial node \( v_0 \). The path in \( G \) that we have so far traversed is stored in the list variable \( \text{current\_path} \). The boolean variable \( \text{satisfied} \) indicates if there is a simple path in \( \text{comp}(N) \) that satisfies \( D \). The set variable \( \text{error\_trace} \) is used to store some subpaths, whose trail include the same event names and numbers as \( D \), but some temporal disorders are detected. The disorder information can be used for further understanding and improvement of the system designs. For each new node discovered, the algorithm first check if the path corresponding to \( \text{current\_path} \) becomes a simple path. If yes, then it checks whether the simple path satisfies \( D \), and assigns \( \text{satisfied} \) with \( \text{true} \). If this simple path do not satisfy \( D \), then the algorithm put the subpath whose trail is a projection over \( D \) into \( \text{error\_trace} \). And if the new node is such that \( \text{current\_path} \) is
not corresponding to a prefix for a simple path, then the algorithm backtracks. Otherwise the algorithm adds the new node to current_path. The algorithm terminates because there is only a finite number of simple paths in G. Since the algorithm is based on depth first search framework, its complexity is proportional to the number and the length of simple paths in the reachability graph.

4.3. Checking forward mandatory existential consistency

For the forward mandatory existential consistency checking problem, which is depicted in Fig.5, the formal definition is given below.

\[
D_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} \xrightarrow{a_i} \ldots \xrightarrow{a_{m-1}} \xrightarrow{a_m} D_2
\]

**Figure 5. Forward mandatory existential consistency**

A comp(N) satisfies the forward mandatory existential consistency specified by two sequence diagrams D_1 and D_2, if any behavior of comp(N) with a form of \( \pi_0 \xrightarrow{a_0} \pi_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} \ldots \xrightarrow{a_{m-1}} \pi_m \) satisfies the following condition:

- if the trail of subsequence \( \pi_i \xrightarrow{a_i} \pi_{i+1} \xrightarrow{a_{i+1}} \ldots \xrightarrow{a_{m-1}} \pi_m \) is an occurrence of D_1, then there is a successive subsequence in comp(N) with a form of \( \pi_{m+1} \xrightarrow{a_{m+1}} \ldots \xrightarrow{a_{m-n}} \pi_n \) (\( m \leq n \)) satisfying that its trail corresponds to an occurrence of D_2 exactly.

To solve the problem basing on the reachability graph G of the comp(N), a forward extended simple path is introduced as follows. Let \( t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_m \rightarrow \ldots \rightarrow t_n \) be a path of G. If it satisfies that the subpath \( t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_m \) is a simple path satisfying D_1 and \( \sigma(t_{m+1}, t_n) \) is a projection over D_2, then we call it a forward extended simple path over D_2. If \( \sigma(t_{m+1}, t_n) \) is a legal projection over D_2 furtherly, then we say that the forward extended simple path satisfies D_2. For any simple path \( \rho = \tau_0 \rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_k \) satisfying D_1, a set of forward extended simple paths over D_2 can be constructed as \( \Theta(\rho) = \{ \tau_0 \rightarrow \tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \tau'_0 = t_i (0 \leq i \leq k) \} \). Theorem 2 Let G be a reachability graph of comp(N). comp(N) satisfies the forward mandatory existential consistency specified by two sequence diagram D_1 and D_2 iff for any simple path \( \rho \) satisfying D_1, there is a path in \( \Theta(\rho) \) which satisfies D_2.

\[
current_{\text{path}} := (\emptyset);
\]

**Figure 6. Algorithm for checking forward mandatory existential consistency**

\[
\begin{align*}
\text{repeat} \\
\text{node} & := \text{the last node of current}_{\text{path}}; \\
\text{if node has no new successive node} & \text{then delete the last node of current}_{\text{path}}; \\
\text{else} & \text{begin} \\
\text{node} & := \text{a new successive node of node}; \\
\text{if node is such that the path corresponding to current}_{\text{path}} \text{is a simple path satisfying } D_1 & \text{then begin} \\
\text{check there is a path in } \Theta(\text{current}_{\text{path}}) & \text{satisfying } D_2; \\
\text{if no return false;} & \text{end} \\
\text{if node is such that the path corresponding to current}_{\text{path}} \text{is a prefix for a simple path satisfying } D_1 & \text{then begin} \\
\text{append node to current}_{\text{path}}; & \text{end} \\
\text{until current}_{\text{path}} = \emptyset; \\
\text{return true.}
\end{align*}
\]

Based on theorem 2, an algorithm is developed to check the forward mandatory existential consistency, which is showed in Fig.6. Notice that it can also record the set of error_trace of any disorder subpaths if necessary, as the same as we have done in the first algorithm. Just for description convenience, the set of error_trace is not be presented explicitly in the following algorithms. Since this algorithm is based on depth first search framework, its complexity is proportional to the number and the length of simple paths and the forward extended simple paths in the reachability graph.

4.4. Checking backward mandatory existential consistency

For the backward mandatory existential consistency checking problem, which is depicted in Fig.7, the formal definition is given below.

A comp(N) satisfies the backward mandatory existential consistency specified by two sequence diagrams D_1 and D_2, if any behavior of comp(N) with a form of \( \pi_0 \xrightarrow{a_0} \pi_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} \pi_i \xrightarrow{a_i} \ldots \xrightarrow{a_{m-1}} \pi_m \) satisfies the following condition:
Figure 7. Backward mandatory existential consistency

- if the trail of subsequence \( \vec{v}_j \xrightarrow{a_j} \vec{v}_{j+1} \xrightarrow{a_{j+1}} \cdots \xrightarrow{a_{m-1}} \vec{v}_m \) is an occurrence of \( D_1 \), then there is a subsequence in \( \text{comp}(N) \) with the form of \( \vec{v}_0 \xrightarrow{a_0'} \vec{v}_1 \xrightarrow{a_1'} \cdots \xrightarrow{a_{i-1}'} \vec{v}_i \), where the tail of \( \vec{v}_j \xrightarrow{a_j'} \vec{v}_{j+1} \xrightarrow{a_{j+1}'} \cdots \xrightarrow{a_{i-1}'} \vec{v}_i \) is an occurrence of \( D_2 \).

To solve the problem basing on the reachability graph \( G \) of the \( \text{comp}(N) \), a set of backward extended simple paths is introduced as follows. Similar to the forward case, for any simple path \( \rho = t_0 \xrightarrow{a_0} t_1 \cdots t_m \) of \( G \) satisfying \( D_1 \), where \( \sigma(t_m, t_n) \) is the occurrence of \( D_1 \), we can construct a set of backward extended simple paths as \( \Delta(\rho) = \{ t_0' \xrightarrow{a_0'} t_1' \cdots t_{m-1}' | t_i = t_m \leq i \leq n \} \). If \( t_0' \xrightarrow{a_0'} t_1' \cdots t_{m-1}' \) is a simple path satisfying \( D_2 \), then we say that there is a path in \( \Delta(\rho) \) satisfying \( D_2 \).

**Theorem 3** Let \( G \) be a reachability graph of \( \text{comp}(N) \), \( \text{comp}(N) \) satisfies the backward mandatory existential consistency specified by two sequence diagram \( D_1 \) and \( D_2 \) if and only if for any simple path \( \rho \) satisfying \( D_1 \), there is a path in \( \Delta(\rho) \) satisfying \( D_2 \).

Based on theorem 3, an algorithm is developed to check the backward mandatory existential consistency, which has the same depth first search framework as that one in the forward case. The only difference is that the \( \Theta(\text{current path}) \) is replaced by \( \Delta(\text{current path}) \). The complexity of this backward checking algorithm is proportional to the number and the length of simple paths and the backward extended simple paths in the reachability graph.

4.5. Checking bidirectional mandatory existential consistency

For the bidirectional mandatory existential consistency checking problem, which is depicted in Fig.8, the formal definition is given below.

A \( \text{comp}(N) \) satisfies the bidirectional mandatory existential consistency specified by the sequence diagrams \( D_1, D_2 \) and \( D_3 \), if any behavior of \( \text{comp}(N) \) with a form of \( \vec{v}_0 \xrightarrow{a_0} \vec{v}_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} \vec{v}_k \xrightarrow{a_k} \cdots \xrightarrow{a_{l-1}} \vec{v}_l \xrightarrow{a_l} \cdots \xrightarrow{a_{m-1}} \vec{v}_m \) where all \( \vec{v}_i \) (\( l \leq i \leq m \)) are distinct, the trail of \( \vec{v}_k \xrightarrow{a_k} \vec{v}_{k+1} \xrightarrow{a_{k+1}} \cdots \xrightarrow{a_{n-1}} \vec{v}_n \) is an occurrence of \( D_1 \), and the trail of \( \vec{v}_m \xrightarrow{a_m} \vec{v}_{m+1} \xrightarrow{a_{m+1}} \cdots \xrightarrow{a_{n-1}} \vec{v}_n \) is an occurrence of \( D_2 \), the following condition holds:

- if there is no subsequence \( \vec{v}_j \xrightarrow{a_j} \vec{v}_{j+1} \xrightarrow{a_{j+1}} \cdots \xrightarrow{a_{j-1}} \vec{v}_j \) (\( l \leq i < j \leq m \)) whose trail is an occurrence of \( D_1 \) or \( D_2 \), then there is a subsequence \( \vec{v}_i \xrightarrow{a_i} \vec{v}_{i+1} \xrightarrow{a_{i+1}} \cdots \xrightarrow{a_{m-1}} \vec{v}_m \) in \( \text{comp}(N) \) whose trail is an occurrence of \( D_3 \) exactly.

Figure 8. Bidirectional mandatory existential consistency

Again, to solve the problem basing on the reachability graph \( G \) of \( \text{comp}(N) \), a compositional simple path is introduced as follows. Let \( t_0 t_1 \cdots t_m \) be a path of \( G \). We call it a compositional simple path satisfying \( D_1 \) and \( D_2 \) if the following conditions hold:

- the subpath \( t_0 t_1 \cdots t_m \) is a simple path satisfying \( D_1 \);
- the subpath \( t_{m+1} t_{m+2} \cdots t_n \) is a simple path satisfying \( D_2 \), and
- \( \sigma(t_{m+1}, t_{n-1}) \) is not any occurrence of \( D_1 \) and \( D_2 \).

Then, for any compositional simple path \( \rho \) with the above form, we can construct a set of compositional simple paths as \( \Omega(\rho) = \{ t_0' t_1' \cdots t_{m-1}' | t_i = t_i(0 \leq i \leq m) \} \). If \( \sigma(t_{m+1}, t_{n-1}) \) is a legal projection over \( D_3 \), then we say that there is a path of \( \Omega(\rho) \) satisfying \( D_3 \).

**Theorem 4** Let \( G \) be a reachability graph of \( \text{comp}(N) \), \( \text{comp}(N) \) satisfies the bidirectional mandatory existential consistency specified by the sequence diagrams \( D_1, D_2 \) and \( D_3 \) if and only if for any compositional simple path \( \rho \) satisfying \( D_1 \) and \( D_2 \), there is a path in \( \Omega(\rho) \) satisfying \( D_3 \).

Based on theorem 4, an algorithm is developed to check the bidirectional mandatory existential consistency. In this case, a path \( \rho \) in \( G \) is called a prefix for a compositional simple path if there could be a subpath \( \rho' \) such that \( \rho' \rho \) is a compositional simple path satisfying \( D_1 \) and \( D_2 \). The main checking framework is mostly as the same as the above two mandatory existential checking algorithms. Only in
the bidirectional case, for each new node discovered, we first check whether it is such that the path corresponding to current_path is a compositional simple path satisfying $D_1$ and $D_2$. If yes, we further check whether there is a path in $\Omega(\text{current_path})$ satisfying $D_3$. Then we check if the new node discovered is such that the path corresponding to current_path is a prefix for a compositional simple path satisfying $D_1$ and $D_2$. The complexity of the algorithm is proportional to the number and the length of compositional simple paths in the reachability graph.

5. Related work and conclusion

In this paper, we use interface automaton networks to model the component-based designs for embedded software systems, and describe the scenario-based specifications by UML sequence diagrams. Based on investigating the reachability graph of the compatible interface automaton networks, we develop the algorithms to check different kinds of existential consistency between the specification and design models. Those interesting properties include not only the existential consistency, but also the mandatory existential consistency, such as: the forward, backward and bidirectional mandatory consistency.

The related works are described as follows. A framework is presented in [10] to extend the concept of type systems in programming languages to capture the interaction in embedded component-based designs. The interaction types and component behavior are also described by the interface automata, and type compatibility can be checked through automata composition. In [9] a modular consistency analysis method for dataflow process networks is presented, where interface automata are employed as a bridge between the architectural model and heterogeneous components representing concrete models of processes. In [5][3], the authors of interface automata further established timed interface automata and resource interface automata, which can be used respectively for describing and checking real-time property and limited resource property of component interface. Other works close to our own also include[8][2]. [8] presents an approach to verify the consistency between statecharts and scenarios models representing the software architecture dynamics and the coordination constrains respectively. The model checker SPIN[6] is also used for verifying the conformance of the implemented behavior with respect to the expected specification. However the temporal requirements of system are not considered. [2] introduces a framework for modelling and specifying the global behavior of web-service compositions which are presented by Mealy machines. A set of conversations which is a sequence of messages interaction can be used to exhibit expected or unexpected behaviors. Compared with above works, our work is a more complete solution for checking behavior compatibility of the component-based embedded software designs for the scenario-based specifications. On the one hand, instead of UML statecharts we use a lightweight automata-based language to describe temporal aspects of embedded software component interfaces, along with modelling compositional designs with interface automaton networks. On the other hand, not only the general existential consistency can be checked, but also the several kinds of mandatory existential consistency can be verified, which are not mentioned in other works.

In this paper, we just consider a simple version of sequence diagrams since our concern is how to check if the interface automata interact according to the scenarios-based specifications. A prototype tool is being developed to implement the solutions presented in the paper. For future work, we are going to consider the alternatives and loops in sequence diagrams. For practical use we need more industrial applications to check and improve the performance of the prototype tool. At the same time, it is necessary to analyze and check sequence diagrams and interface automata with real-time constrains.

References