Bản quyền thuộc Tạp chí CNTT&TT

A Formal Method for Specifying The Interface of Components in Real-time Concurrent Systems

Do Van Chieu
Faculty of Information Technology, HaiPhong Private University
Email: chieudv@hpu.edu.vn

Abstract - **In this paper, we propose a method for specifying the interface of components in real-time concurrent systems. The key idea of the proposed method is to extend Interface-based design with using timed trace theory. We propose a technique to specify the interaction protocols of component interfaces by the languages of timed words augmented with the concurrency, i.e. timed trace languages. In addition, we propose a class of automata that can recognize a class of timed trace languages called timed concurrent interface automata. We give an algorithm for the refinement, component composition, and show that our method possesses two important features of interface automata theory which are incremental design and independent implementation. Those results play a key role in the specification and verification of real-time concurrent systems.**

Keywords* - *concurrent systems, Mazurkiewicz trace, linear temporal logic, timed trace, duration trace, Asynchronous duration automata, interface automata.

1. INTRODUCTION

The component-based development of real-time systems is considered as an efficient approach for developing real-time systems because of the reductive time and low cost while retaining the software quality. According to this method, a complex system is made of components. All individual components are packages, web services or software modules. They are connected to each other via interfaces. One of the major challenges of this approach is how to ensure that the composition of components is valid and that the resulting system meets its requirements. To deal with this problem, formal methods have been proved to be more efficient. Interface-based design [3, 13] is such a typical method. Based on this method, each component of a system is specified by a tuple of Input/Output ports and external behaviors, not internal behaviors. Because the system specification is formal, the behaviors and requirements of the system can be specified exactly, and therefore we can apply formal verification techniques to prove the correctness and many valid properties of the system by using automatic or semi-automatic tools.

In real-time concurrent systems, apart from the aspects mentioned as above, there are execution-time constraints that they need to satisfy, and they may have some parts running in parallel for an efficient implementation. Hence, the methods mentioned above may not be strong enough for specifying and verifying real-time concurrent systems. To solve the problem, several methods have been proposed, but they still have some limitations. Recent researches [18, 7 - 9] proposed several methods for specifying and verifying real-time concurrent systems. These methods usually use timed automata [4] and similar techniques to specify components. So each component is modeled by a timed automaton. However, specifying the concurrency by timed automata is difficult and complicated. To overcome this problem, some other methods [14, 15, 25, 27] have been proposed in order to support the specification of concurrency. However, they do not support specifying time constraints. Some others can specify real-time concurrent executions [7] but have not support component based systems. Therefore, searching a good technique for specifying and verifying the correctness and validity of component-based real-time systems is still an attractive topic in software technology.

In this paper, we propose a method to specify component based real-time systems based on the interface theory by extending it with timed and concurrent protocols in order to support real-time concurrent system specification. We suppose that an action of a system includes functional specification, non-functional specification and worst case execution time. So interaction protocols in component interface need to satisfy three following constraints:

Bản quyền thuộc Tạp chí CNTT&TT

1. The sequencing constraints: the interactive actions should obey some constraints on the order they occur
2. The timing constraints: there are many kinds of time constraints. In a component, the most critical constraints are those saying that the services (methods) cannot be called so frequently if they cannot be executed in parallel. This means that there should be some minimum time distance between the actions that must be sequencing.
3. The constraints on the parallel calls from different threads: Which services can be called in parallel with which ones.

To conduct this research, we propose to use interface automata, which can recognize languages with timed words and concurrent constraints. Hence, the interface interaction protocol of each component is a timed trace language recognized by timed interface automaton. The contribution in this paper is to give the composition, refinement and to show two aspects of the component based development, which are incremental design and independent implementation.

The rest of the paper is organized as follows: The next section introduces the theory of timed trace and asynchronous duration automata. These theories are developed to support the verification and specification of systems that have non-functional requirements. Sections 3 presents some techniques for checking the compatibility, for composition and refinement. Section 4 discusses the specification technique for real-time concurrent systems. Finally, Section 5 is the conclusion of the paper.

2. TIMED TRACE AND ASYNCHRONOUS DURATION AUTOMATA

Times trace and asynchronous duration automata had been proposed in [7, 8, 9]. These studies have shown the benefits of timed traces to support the specification of real time concurrent systems. Such benefits include the simplicity and the precision of representation of the system behaviors in the form of automata or a linear time logic formula. In this section, we recall some concepts and important results that will be used in this paper.

2.1.Timed traces

A dependence alphabet is a pair (Σ, D) where Σ is a finite alphabet, D is a binary reflexive and symmetric relation on Σ and is called dependence relation. Given D, the independence relation I is the complement of D. We call (Σ, I) independence alphabet. For a set $A \subseteq \Sigma$, the set of letters dependent on A is denoted by $D(A) = \{b \in \Sigma \vee (a,b) \in Dforsomea \in A\}$. A Mazurkiewicz's trace is an isomophic class of a labeled partial order $T = (V, \leq, \lambda)$ where V is a set of vertexes labeled by $\lambda: V \rightarrow \Sigma$ and $\leq$ is a partial order over V satisfying the following conditions:

- For all $x \in V$, the downward set $\downarrow x = \{y \in V \vee y \leq x\}$ is finite (and we call it the history of event x), and
- for all $x, y \in V$ we have that $(\lambda(x), \lambda(y)) \in D$ implies $x \leq y$ or $y \leq x$, and that $x \lessdot y$ implies $(\lambda(x), \lambda(y)) \in D$, where $\lessdot \widehat{=} \leq \setminus \leq^2$.

As usual, Σ^* and Σ^ω denote the set of finite and infinite words over Σ respectively, and Σ^∞ denotes $\Sigma^* \cup \Sigma^\omega$. A word in Σ^∞ is associated with a trace over (Σ, D) by the mapping $wtot: \Sigma^\infty \rightarrow Tr(\Sigma, D)$ defined as: for $\omega \in \Sigma^\infty, wtot(\omega)$ is (the equivalence class of) $\langle V, \leq, \lambda \rangle$where:

- $V = pref(\omega) - \{e\}$,
- $\leq$ is the least partial order over V satisfying that for $\tau a, \tau' b \in V$ if τa is a prefix of $\tau' b$ and if $(a, b) \in D$ then $\tau a \leq \tau' b$, and
- $\lambda(\tau a) = a$.

We define the mapping $ttow: Tr(\Sigma, D) \rightarrow \Sigma^\omega$ as $ttow([V, \leq, \lambda]) \widehat{=} \{\lambda(\delta) \vee \delta$ is a linearization of $(V. \leq)\}$. The map $ttow$ is extended to be defined on trace languages as follows: for any trace language L over (Σ, D), $ttow(L) \widehat{=} \bigcup_{T \in L} ttow(T)$[7].

Now, we introduce some notions about the timed traces as an extension of the traces. Let time be continuous and represented as the set of non-negative real $R^{\geq 0}$. Let $\leq$ also represent the natural ordering in $R^{\geq 0}$ without the fear of confusion since its meaning is clear from the context. As for the case of words, we add a labeling function θ to associate a vertex of a trace with a time point in $R^{\geq 0}$

Definition 1 (Timed Trace) *A timed trace over (Σ, D) is a pair (T, θ) where*

Bản quyền thuộc Tạp chí CNTT&TT

- $T = (V, \leq, \lambda)$ *is a trace over* (Σ, D),
- $\theta: V \to R^{\geq 0}$ *satisfying:*
  - $e < e' \Rightarrow \theta(e) \leq \theta(e')$ *(time should respect the causality), and*
  - *if* T *is infinite, for any* $t \geq 0$, *there is a cut* C *of* T *such that* $min\{\theta(e) \vee e \in C\} \geq t$ *(time should be progress and divergent)*

A set of timed traces over (Σ, D) is called a timed trace language [7]. Let $in\ tv$ be the set of all time durations over $R^{\geq 0}$, $i\ ntv \widehat{=} \{[l, u] \vee l \in R^{\geq 0} \wedge u \in R^{\geq 0} \cup \{\infty\}$. Let $J_i: \Sigma_i \to intv$ be a function that associates a time duration to each $a \in \Sigma$, and $J(a) \widehat{=} (J(a))_{j \in lo\ c(a)}$ with $loc(a) = \{i \in Proc | a \in \Sigma_i\}$. $J(a)$ could be interpreted as a time constraint for the execution time of the action a in each process that involved in a.

Definition 2 (Duration Trace): *Given* $J: \Sigma \to intv$ *and a trace* $T = (V, \leq, \lambda)$.

- *The pair* (T, J) *is called an duration trace.*
- *The timed trace language defined by the duration trace* (T, J), *denoted by* $ttr(T, J)$ *is defined as* $\{(T, \theta) \vee (T, \theta)$ *is a timed trace and* $\forall e \in V, \forall e' \in <_e, \lambda(e') \in \Sigma_i \Rightarrow \theta\ (e) - \theta e' \in J_i(\lambda(e))\}$.

Let $ttr(T, J)$ be $\{(T, \theta) \vee (T, \theta) satisfies (T, J)\}$. Given the interval dependence alphabet (Σ, D, J) and a trace language *L* over (Σ, D), we define timed trace language $tTr(L, J)$ as $t\ Tr(L, J) \ \widehat{=} \cup_{T \in L}\ tt\ r(T, J)$

Example 1 *Given* $T = \langle V, \leq, \lambda \rangle$ *is a trace over* $D = \{a, b\}^2 \cup \{a, c\}^2$ *where* $\Sigma = \{a, b, c\}, V = \{e_1, e_2, e_3, e_4, e_5\}$, *partial order* $\leq$ *defined as:* $e_1 \leq e_2, e_2 \leq e_3, e_3 \leq e_4, e_4 \leq e_5$, $\lambda(e_1) = a, \lambda(e_2) = b, \lambda(e_3) = c, \lambda(e_4) = b, \lambda(e_5) = a$, *function* J: $J(a) = [1,2], J(b) = [2,4], J(c) = [1,3]$. *A timed trace* (T', θ) *defined by a duration trace* (T, J) *is shown in Figure 1.*

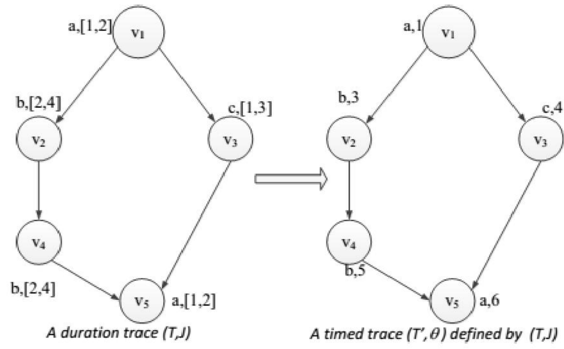

Figure 1*: A timed trace defined by a duration trace in Example 1*

2.2. Asynchronous Duration Automata

As in [17] we call $\tilde{\Sigma} = \{\Sigma_1, \ldots, \Sigma_m\}$ a distributed alphabet, and and $\tilde{\Gamma} = \{\Gamma_1, \ldots, \Gamma_m\}$ a distributed interval alphabet where $\Gamma_j = \{(a, J(a)) | a \in \Sigma_j\}$. Let $\Gamma = \cup_{j \in Pr\ oc} \Gamma_j\}$. In the sequel we use the following notations. For $\omega \in \Sigma^{\omega}, proj_i(\omega)$ denotes the projection of the word ω on Σ_i; and $pref(\omega)$ denotes the set of all prefixes of ω. Let us define $loc(a) = \{j \vee a \in \Sigma_j\}$ for any $a \in \Sigma$. For a set $\{S_j\}_{j \in Proc}$, and $a \in \Sigma$ with $loc(a) = \{j_i, j_2, \ldots, j_k\}$we denote by S_{Proc} the Cartesian product $\prod_{j \in Proc} S_j$, and by S_a the Cartesian product $S_{j1} \times S_{j2} \times \ldots \times S_{jk}\}$.

Definition 3: *An asynchronous automaton over* $\tilde{\Sigma}$ *is a structure* $\mathcal{A} = (\{S_i\}_{i \in Proc}, \{\to_a\} a \in \Sigma, S_{in}, \{F_i\}_{i \in Proc}, \{G_i\}_{i \in Proc})$ *where:*

- *Each* S_i *is a finite set of i-local states,*
- $\to_a \subset S_a \times S_a$ *for each a is a set of a-transitions, and*
- F_i, G_i *are subsets of* S_i *for each* $i \in Proc$.

Let J defined as above, an asynchronous duration automaton is an asynchronous automaton is equipped with a timed mapping $J$with $J(a)(= (J_i(a))_{i \in loc(a)})$ for each transition a $(a - transition)$.

Definition 4 (Asynchronous Duration Automata) *An Asynchronous Duration Automaton is a pair* (A, J) *where* A *is an asynchronous duration automaton.*

We now define when a timed word is accepted by (A, J) directly to justify our interpretation of time constraints for interval traces.

Bản quyền thuộc Tạp chí CNTT&TT

A run on a timed word $\omega \in (\Sigma \times R^{\geq 0})^{\omega}$ is a map $\rho: pref(\omega) \rightarrow S_{Proc}$ defined by:

- $(\varepsilon) \in S_{in}$, and
- for all prefix $\tau(a,t)$ of ω, $\rho(\tau) \xrightarrow{a}_A \rho(\tau a)$ and $t - time_i(\tau) \in J_i(a)$ for all $i \in loc(a)$ where for a time word $\tau = \omega'(b,t')\omega''$ such that $b \in \Sigma$ and ω'' has no occurrence of a symbol in Σ_i, we define $time_i(\tau) \widehat{=} t'$.

The run ρ is an accepting run iff for each $j \in Proc$ either

- $Proj_j(\omega)$ is finite, and $\rho(\omega')(j) \in F_j$, where $\omega' \in Pref(\omega)$ and $Proj_j(\omega') = Proj_j(\omega)$, or
- $Proj_j(\omega)$ is infinite and $\rho(\tau)(j) \in G_j$, for infinitely many $\tau \in pref(\omega)$.

When ρ is an accepting run on timed word ω we say that ω is accepted by (A,J). The set of all timed words accepted by asynchronous duration automaton (A,J) is called timed language accepted by (A,J) and denoted by $tL(A,J)$. Like for the untimed case [24], we have:

Theorem 1 $tL(A,J) = \bigcup_{T \in TrL(A)} tword(ttr(T,J))$.

The following definition gives a timed trace language accepted by asynchronous duration automaton.

Definition 5: *Timed trace language accepted by an asynchronous duration automaton (A,J) is defined as $tTrL(A,J) \widehat{=} \bigcup_{T \in TrL(A)} ttr(T,J)$.*

If a word $\omega \in \Sigma^{\omega}$ is accepted by A then any word $\omega, ttow(wtot(\omega))$ is accepted by A. We define the trace language accepted by A as $TrL(\mathcal{A}) \widehat{=} wtot(L_{sym}(\mathcal{A}))$.

We have following result according to these results in [7, 9] about empty checking problem of asynchronous duration automaton.

Proposition 1: *Let (A,J) is an asynchronous duration automaton over (Σ, D), the empty checking problem of (A,J) is decidable.*

4. TIMED CONCURRENT INTERFACE AUTOMATA

From these above results, a timed trace can hold all three characteristics of the real-time concurrent protocol mentioned in the introduction. Moreover, the duration alphabet can be represented constraints on the implementation of the actions of the system. We can use the finite asynchronous automaton A to represent a trace language L, and extend it with a duration function $J: \Sigma \rightarrow intv$ to represent a timed trace language. Note that the Marzurkiewicz trace languages are very efficient for simultaneous binding, we will use a set of timed traces that has a finite representation as a specification for the interfaces (in [8]). In this section, we provide a specification method for concurrent real-time interfaces.

A concurrent real-time system is composed of real-time concurrent components. Each real-time concurrent component has an interface with its interaction protocol satisfying the three constraints introduced in Section 1. Therefore, ADAs with their recognized timed trace languages representing component interaction protocols are suitable for the specification of the system components. In this section, we will use these automata with some constraints on actions set for specifying interface of components. We also give concepts of composition conditions, compatible, refinement and method for parallel composition of components.

3.1. Definitions

As introduced in Section 1, the protocol of a real-time concurrent component interface is timed trace language. For the finite representation of the languages, we use ADA. Therefore, a component interface protocol can be specified as a ADA. A timed concurrent interface automata is a ADA with input and output actions of system. We give a formal definition as follows.

Definition 6 (Timed Concurrent Interface Automata): *A timed concurrent interface automaton (denoted by TCIA) is a 3-tuple $P = (I, O, (A^D, J))$, where I is a set of input actions, O is a set of output actions and $(A^D, J) = (\{S_i\}_{i \in Proc_A}, \rightarrow_{\mathcal{A}}, S_{in}, \{F_i\}_{i \in Proc_A}, \{G_i\}_{i \in Proc_A})$ is deterministic asynchronous duration automaton with the alphabet $\Sigma = I \cup O$.*

For the sake of simplicity but without loss of expressiveness, only deterministic asynchronous duration automata are used in modeling interface and we denote A in A^D instead of A^D.

Given a TCIA P, the interface language of P is defined as the language of ADA (A,J), i.e, the

Bản quyền thuộc Tạp chí CNTT&TT

language of P is a timed trace language recognized by ADA (A, J). We denote set of state transition of TCIA P as $tran(P) = tran(A, J) = \rightarrow_A$. If $a \in I$ $(a \in O)$ then $(s, a, s') \in tran(A, J)$ is called input (output) transition of TCIA P and denoted $tran^I(P)$ $(tran^O(P))$.

An action $a \in \Sigma$ is called to active at state $s \in S_{Proc}$ if $(s, a, s') \in tran(P)$. We denote $\Sigma(s) = I(s) \cup O(s)$ as a set of all actions activated at s. All input actions in I\I(s) is set of unacceptable at state s.

Example 2: *Give TCIA* $P = (I_P, O_P, (A_P, J_P))$ *where* $I_P = \{a\}, O_P = \{b, c\}, Proc_P = \{1,2\}, \tilde{\Sigma} = \{\{a, b\}, \{a, c\}\}, S_P^{in} = \{q_{10}, q_{20}\}, F_P = G_P = \{q_{11}, q_{21}\}, S_P = \{q_{10}, q_{20}, q_{11}, q_{21}\}, J_P(a) = [1,2], J_P(b) = [2,3], J_P(c) = [1,3]$, *and independent relation* $I_P = \{(b, c), (c, b)\}$, *because* b *and* c *belong to 2 different processes. Timed trace language of* P *is set of timed traces such that they satisfy duration trace* $T = \{(a(bca)^\omega, J_P)\}$. *A presentation of* P *is shown in Figure 2, these action transitions are shown in Table 1.*

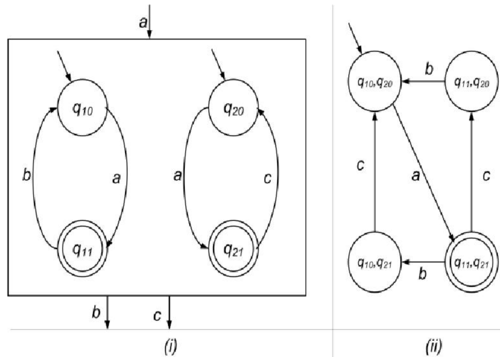

Figure 2: A TCIA P *where* $J_P(a) = [1,2], J_P(b) = [2,3], J_P(c) = [1,3]$ *(i) and it's state transition graph (ii)*

Table 1. A transition table of TCIA P *in Example 2*

(q_{10}, q_{20})	$\xrightarrow{a}$	(q_{11}, q_{21})
(q_{10}, q_{21})	$\xrightarrow{c}$	(q_{10}, q_{20})
(q_{11}, q_{20})	$\xrightarrow{b}$	(q_{10}, q_{20})
(q_{11}, q_{21})	$\xrightarrow{b}$	(q_{10}, q_{21})
(q_{11}, q_{21})	$\xrightarrow{c}$	(q_{11}, q_{20})

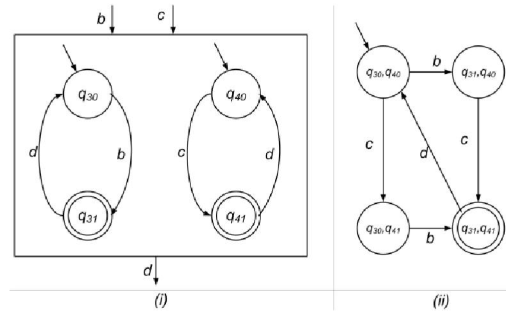

Figure 3: TCIA Q *where* $J_Q(b) = [2,3], J_Q(c) = [1,3], J_Q(d) = [2,4]$ *(i) and it's state transition graph (ii) is compatible with TCIA* P *in Example 2*

3.2. Composability and Parallel Composition of TCIA

According to Interface Automata theories, we need to give a method for composition between TCIAs to build a big component which has more functions. So, given two automata P and Q, constructing their composition is give a TCIA which specifies the composite interface of them $R = P \vee Q$. Two TCIA P and Q is composable if the composition between them is not empty. Firstly, we give a definition for Composability of automata.

Definition 7 (Composability): *Two TCIA* P *and* Q *are composable if*

1. *Set of input actions* I_P *and* I_Q *and set of output actions* O_P *and* O_Q *is not intersection,*
2. *These timed functions* J_P *and* J_Q *over shared actions* $shared(P, Q) = \Sigma_P \cap \Sigma_Q$ *is not conflict, i.e. for all action* $a \in shared(P, Q), J_P(a) \cap J_Q(a) \neq \emptyset$*, and*
3. *Set of processes* $Proc_P$ *and* $Proc_Q$ *is not intersection.*

Definition 8: *Given two composable TCIAs* P *and* Q*, the set of illegal states denoted* $Illegal(P, Q) \subseteq S^P \times S^Q$ *where* $S^P = \{S_i\}_{i \in Proc_P}$ *and* $S^Q = \{S_i\}_{i \in Proc_Q}$ *is a set of states of* P *and* Q *respectively, defined as following:*

$Illegal(P, Q) = \{(s^P, s^Q) \in S^P \times S^Q$ *such that exists* $a \in shared(P, Q)$ *where* $a \in O_P(s^P) \wedge a \notin I_Q(s^Q)$ *or* $a \in O_Q(s^Q) \wedge a \notin I_P(s^P)\}$

Bản quyền thuộc Tạp chí CNTT&TT

Example 3: *Given a TCIA* $Q = (I_Q, O_Q, (A_Q, J_Q))$ *vá»›i* $I_Q = \{b, c\}, O_P = \{d\}, Proc_Q = \{3,4\}, \tilde{\Sigma}_Q = \{\{b,d\},\{c,d\}\}, S_Q^{in} = \{q_{30}, q_{40}\}, F_Q = G_Q = \{q_{31}, q_{41}\}, S_P = \{q_{30}, q_{40}, q_{31}, q_{41}\}, J_Q(d) = [2,4], J_Q(b) = [2,3], J_Q(c) = [1,3]$ *and the independent relation* $I_Q = \{(b,c),(c,b)\}$ *because* b *and* c *belong to 2 different processes. The timed trace language of* Q *is set of timed traces such that they satisfy duration trace* $T = \{((bcd)^\omega, J_Q)\}$. *A presentation of* Q *is shown in Figure 3.*

***Table 2**. The transition Table of TCIA* Q *in Example 3*

(q_{30}, q_{40})	$\xrightarrow{c}$	(q_{30}, q_{41})
(q_{30}, q_{40})	$\xrightarrow{b}$	(q_{31}, q_{40})
(q_{30}, q_{41})	$\xrightarrow{b}$	(q_{31}, q_{41})
(q_{31}, q_{40})	$\xrightarrow{c}$	(q_{31}, q_{41})
(q_{31}, q_{41})	$\xrightarrow{d}$	(q_{30}, q_{40})

Now, we are ready to give definition about parallel composition between two TCIAs

Definition 9 (Parallel Composition): *The parallel composition of composable TCIAs* $P = (I_P, O_P, (A_P, J_P))$ *and* $Q = (I_Q, O_Q, (A_Q, J_Q))$ *denoted by* $P \parallel Q$ *is a TCIA* $F = (I_F, O_F, (A_F, J_F))$ *where:*

- $O_F = O_P \cup O_Q$ and $I_F = (I_P \cup I_Q) \setminus O_F$,
- $Proc_F = Proc_P \cup Proc_Q$, $\tilde{\Sigma}_F = \tilde{\Sigma}_P \cup \tilde{\Sigma}_Q$, $J_F = J_P \uplus J_Q = \{J_P(a) \cup J_P(b) | a \in \Sigma_P, b \in \Sigma_Q, a \neq b\} \cup \{J_P(a) \cap J_Q(a) | a \in \Sigma_P \cap \Sigma_Q\}$, and
- $A_F = (\{S_i\}_{i \in Proc_P} \times \{S_i\}_{i \in Proc_Q}, \{\xrightarrow{a}_F\}_{a \in \Sigma_F}, \{(s^P, s^Q) | s^P \in S_{in}^P \wedge s^Q \in S_{in}^Q\}, \{F_i\}_{i \in Proc_P} \times \{F_i\}_{i \in Proc_Q}, \{G_i\}_{i \in Proc_P} \times \{G_i\}_{i \in Proc_Q})$, where $\xrightarrow{a}_F = \{(s_a^P, s_a^Q) \xrightarrow{a}_F (s'^P_a, s'^Q_a) | s_a^P \xrightarrow{a}_P s'^P_a \wedge s_a^Q \xrightarrow{a}_Q s'^Q_a \wedge a \in (\Sigma_P \cap \Sigma_Q)\} \cup \{(s_a^B, s_a^C) \xrightarrow{a}_F (s'^B_a, s_a^C) | s_a^B \xrightarrow{a}_B s'^B_a \wedge a \in (\Sigma_B \setminus \Sigma_C)\} \cup \{(s_a^B, s_a^C) \xrightarrow{a}_F (s_a^B, s'^C_a) | s_a^C \xrightarrow{a}_C s'^C_a \wedge a \in (\Sigma_C \setminus \Sigma_B)\}$.

From the definition above, because the parallel composition of two TCIAs is also a TCIA, so the emptiness problem of parallel composition is decidable (based on Proposition 1)

Example 4: *The parallel composition of TCIA* P *and* Q *is shown in Figure 4, and its accepted timed trace language is set of timed traces that satisfy duration trace* $T = \{(a(bcad)^\omega, J\}$ *where* $J = J_P \cup J_Q$ *and independent relation* $I = \{(a,d),(b,c)\}$.

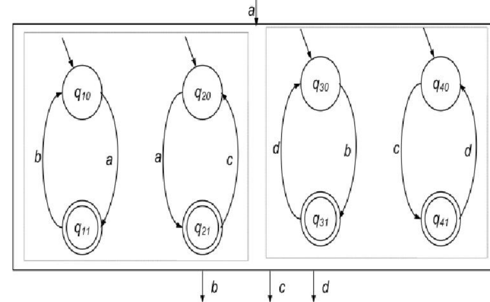

Figure 4: The result of parallel composition of P *and* Q *in Figure 2 and 4*

The problem is that in the composition automaton can exist the illegal states (due to the provision of services by each automaton is missing or conflict) or infertility states. We need to give solution to prevent these states. The general solution for this problem is to provide an environment for composition. Hence, an environment is a TCIA that provide enough conditions for component composition.

Definition 10 (Environment): *An environment for a TCIA* P *is a TCIA* E *that can satisfy these conditions as follow:*

- E *and* P *are composable,*
- E *not empty,*
- $I_E = O_P$*, and*
- $Illegal(P, E) = \emptyset$.

Example 5: *TCIA* Q *in Figure 4 is an environment of TCIA* P *shown in Figure 2.*

Similar to un-timed case, we also have concept of compatibility of two composable TCIAs.

Definition 11 (Compatibility): *Two TCIAs* P *and* Q *are compatible if they are non-empty, composable and the composite automaton is not empty.*

From the parallel composition, given composable TCIAs P, Q and R, the association property of them is still satisfied. It is expressed through the following theorem.

Theorem 2: $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$

Proof: Easily to proof according to definition.

An interface automaton represents assumptions about the environment and guarantees of the

Bản quyền thuộc Tạp chí CNTT&TT

component. Output steps encode assumption that the output must be accepted. Unaccepted input actions at a state prohibits the environment from providing that input. Guarantees about sequence and choice of actions. Composition combines both environment assumptions and component guarantees. There is a particularly simple legal environment for every composable P and Q. It accepts all outputs of $P \parallel Q$, does not provide any inputs. This environment avoids entering illegal states whenever possible.

3.3. Refinement

When developing component-based systems, we need to care about making components to provide more services and require less from their environment. So, at each state, we have concepts of state refinement to have a state which provides more outputs and requires less inputs. We give the following formal definition.

Definition 12 (State Refinement): *Given two TCIAs P and Q, a state refinement relation from P to Q is a binary relations $\pm \in S^P \times S^Q$ such that for all $(u, v) \in S^P \times S^Q, v \pm u$ the following conditions must be satisfied.*

- $I_P(u) \subseteq I_Q(v)$,
- $O_Q(v) \subseteq O_P(u)$,
- $\forall a \in \left(I_P(u) \cap I_Q(v)\right) \cup \left(O_P(u) \cap O_Q(v)\right), J_Q(a) \subseteq J_P(a)$, *and*
- $\forall a \in O_Q(v) \cup I_P(u)$ *and for all state* $u' \vee (u, a, u') \in tran(P)$ *there exist a state* $v' \vee (v, a, v') \in tran(Q)$ *such that* $v' \pm u'$.

From this definition, we now give a concept of interface refinement.

Definition 13 (Interface Refinement): *A TCIA Q is called refinement from TCIA P and denoted by $Q \pm P$ if:*

- $I_Q \subseteq I_P$,
- $O_P \subseteq O_Q$, *and*
- *there exist a state refinement relation from P to Q such that if $u \in S_{in}^P, v \in S_{in}^Q$ then $v \pm u$.*

So, a TCIA always refined from itself, i.e. given a TCIA P, we have $P \pm P$. Obviously, we have the following results which are deduced directly from the definition of the refinement.

Theorem 3: *Given 3 TCIAs P, Q, R, if $P \pm Q$ and $Q \pm R$ then $P \pm R$.*

Finally, our results indicate an important role in ensuring the independent implementation of the theory of interface language for our model. This result indicates the relationship between the parallel composition and refinement of TCIAs. That is, if two different TCIA is refined, then their parallel composition with another TCIA is refinement performance. This ensures that a TCIA is coupled to the system, it also refined TCIA can be composed into the system.

Theorem 4: *Given 3 TCIAs P, Q and R such that Q and R are composable and $I_Q \cap O_R \subseteq I_P \cap O_R$. If P and R are compatible and $Q \pm P$ then Q and R are compatible and $Q \parallel R \pm P \parallel R$.*

Proof: We show that the composition automata $Q \parallel R$ and $\pm P \parallel R$ satisfy three conditions of refinement according to Definition 13.

1. We show that $I_{Q\parallel R} \subseteq I_{P\parallel R}$. From the assume $Q \pm P$ we have $I_Q \subseteq I_P$, hence $I_{Q\parallel R} \subseteq I_{P\parallel R}$.
2. We show that $O_{Q\parallel R} \subseteq O_{P\parallel R}$. From the assume $Q \pm P$ we have $O_Q \subseteq O_P$, so $O_{Q\parallel R} \subseteq O_{P\parallel R}$.
3. There exists a state refinement relation from $P \parallel R$ to $Q \parallel R$ such that $u \in S_{in}^{P\parallel R}, v \in S_{in}^{Q\parallel R}$ then $v \pm u$. Because of $Q \pm P$, there exists a state refinement relation from input state of P into input state of Q. Furthermore, input states of $P \parallel R$ include of input states of P and R, input state of $Q \parallel R$ include of input states Q and R and $R \pm R$. Therefore, from the state refinement definition we have $v \pm u$ where $u \in S_{in}^{P\parallel R}, v \in S_{in}^{Q\parallel R}$.

Consequently, in the interface-based design, a system will be specified by composing many interfaces of components. According to this model, a timed concurrent system is specified by a timed concurrent interface automaton. This automaton is the result of the parallel composition of interface components.

4. RELATED WORKS

At present, there have been many languages supporting the specification of the interface of components, but these languages are either informal or complicated in modeling systems. Therefore, recently researches focus on formal method to design systems based on the interface theory with less complex and more efficiency. In this section, we overview some

Bản quyền thuộc Tạp chí CNTT&TT

methods relating to interface automata theory and methods for modeling concurrent systems.

Luca de Alfaro and Thomas A. Henzinger who established fundamental notions of interface theory and used it for specifying components [13]. In this method, each component considers as a "BlackBox", users only know preconditions and post-conditions of components (called interface), hence each component is represented by its interface and specified by an interface automaton. Systems will be built by composing components based on component operators that are defined. This paper focuses on the ways, which are interactive between components, composition, and composes condition of components. The results have been applied for extending theory and practice. Furthermore, some methods are proposed by Laurent Doyen [16] who arguments interface theory with reuse feature for specifying components. Stavros Tripakis and partners [26] extend the work of De Alfaro, Henzinger et al, on interface theories for component-based design where such input-output relations can be captured. This theory supports both stateless and stateful interfaces, includes explicit notions of environments and pluggability, and satisfies fundamental properties such as preservation of refinement by composition, and characterization of pluggability by refinement.

Recently, there exist studies and methods proposed using interface automata theory to give method for specification and verification component-based systems. In [5], Angelov introduced a method for specification embedded control system by using interface automata. Chouali and colleagues [10] proposed a formal method for verifying component assembly. Cao and partners in [6] extended interface automata with z notation in order to give a specification approach combining interface automata and Z language [2]. Another application using interface automata proposed by Li and his colleagues in [21]. In this study, the authors presented a new web services composition model and its verification algorithm based on interface automata and extended this automata to supports semantic descriptions of web services. Aarts in [1] studied and gave a framework for history dependent abstraction learning to get rid of his previous frameworks using interface automata theory. Lüttgen in [22] introduced a method for modifying model interface automata (MIA) to deal with internal computations and studying a MIA variant make interface automata with optimistic and pessimistic compatibility. This automaton is called richer interface automata.

The biggest limitation of these methods is not to support specifying timed constraints. To dealing with the timed constraints problem, Alur and Dill in [4] gave a theory of timed automata that becomes a fundamental and powerful modeling technique for the development of real-time systems. However the class of timed words languages accepted by timed automata is not closed for complement operator and it is not easy to model distributed systems by (a network of) timed automata (with UPPAAL model checker[1]). The results in [18] are proposal to specify component based real-time systems. They have given a notion about real-time interface automata depending on timed automata in order to specify interface-based systems. These methods also supply refining, checking validation and composing. However, they are still limitations which are not specification concurrent constraints over components. D.V.Hung and Truong Hoang in [12] proposed a method that is a timed extension of relation interface theory of Stavros Tripakis and partners. They introduced the concept of Real-time interfaces which are interfaces with timing constraints relating the time of outputs with the time of inputs. However, this method does not support specifying concurrent systems.

The results in [19, 20, 23, 28] have launched formal methods for the specification and verification of concurrent systems based on Mazurkiewicz's traces. However, their methods focus only on systems with no timing constraints. To solve this problem, D.V. Chieu and D.V. Hung have proposed timed trace theory, which extended timed feature in trace theory [7]. The authors indicated the benefit of this theory for specifying real-time concurrent systems in which targets flexible representation, concision, etc. Especially, this theory deals with two important aspects, they are finite representation by asynchronous duration automata and LTL over timed trace. The paper explicitly indicates timed trace more flexibly represents than timed language and timed automata. Depending on the results in [8], authors apply timed trace theory for rCOS[27] in specifying component based systems with real-time concurrent feature. However, these results do not care about input/output requirements (A new approach solves problem, which is designer merely takes care of external constraint of components but do not take care internal behaviors and consider components as a "BlackBox"). The other

[1] www.uppaal.org/

an application for specification is real-time distributed systems has proposed in [9] but do not mentioned interface automata so it cannot apply for specifying component based real-time systems. Studies in [11] used timed traces in solving runtime verification problems for real-time systems. Runtime verification is checking whether a system execution satisfies or violates a given correctness property. A procedure that automatically, and typically on the fly, verifies conformance of the system's behavior to the specified property is called a monitor. The results of this study offer a technique using two "black boxes", the system and its reference model, are executed in parallel and stimulated with the same input sequences; the monitor dynamically captures their output traces and tries to match them.

In brief, the above results either only deal with a piece of specification problem or propose a method, but complicate or do not suit and difficult to deploy on real-time concurrent systems. In real-timed concurrent systems, the paper uses real-time interface automata can comprehensively solve problem because of its illustration, simple and reliability.

5. CONCLUSION

The paper has proposed a method to specify real-time concurrent systems. The main idea of this method is to use the interface theory and the timed trace theory as the foundation for the method. According to this method, each component of a system is specified as timed concurrent interface automaton. Those automata for components are asynchronous duration automata which have the set of actions separated into two non-empty sets, one is input actions set and the other is output actions set.

In our work, we define the compatibility properties, the composition of components, environments, and the refinement of components. The method guarantees two basic features of the interface based design theory that are incremental design and independent implementation. All above features ensure that a system can be extended with composite compatible components independently on the order of composition (association properties). Besides, a system can be improved by composing the refinements from old ones in order to make a new system, which can support better services at output, but needs least requirements in input. However, the mentioned method has not supported specifying by Logic and is missing a tool for model checking. In the future, we will complete the study by adding specifying systems based on logic in context of Linear Temporal Logic over Timed Trace. Therefore, a system can be verified in design step with existing specification tools.

ACKNOWLEDGMENT

This work is supported by the project no. QG.13.01 granted by Vietnam National University, Hanoi (VNU).

REFERENCES

[1]. Aarts, F. Heidarian, and F. Vaandrager. A theory of history dependent abstractions for learning interface automata. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, pages 240–255, Berlin, Heidelberg, 2012. Springer-Verlag.

[2]. Abrial. Data semantics. In *IFIP Working Conference Data Base Management*, pages 1–60, 1974.

[3]. D. Alfaro and T. A. Henzinger. Interface-based design. In *In Engineering Theories of Software Intensive Systems, proceedings of the Marktoberdorf Summer School*. Kluwer, 2004.

[4]. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[5]. Angelov, F. Zhou, and K. Sierszecki. Specification of embedded control systems behaviour using actor interface automata. In *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems*, SEUS'10, pages 167–178, Berlin, Heidelberg, 2010. Springer-Verlag.

[6]. Cao and H. Wang. Extending interface automata with z notation. In *Proceedings of the 4th IPM International Conference on Fundamentals of Software Engineering*, FSEN'11, pages 359–367, Berlin, Heidelberg, 2012. Springer-Verlag.

[7]. V. Chieu and D. V. Hung. An extension of mazukiewicz traces and their applications in specification of real-time systems. In *Proceedings of the second international Conference on knowledge and systems engineering 2010*. IEEE Computer Society, 2010.

[8]. V. Chieu and D. V. Hung. A formal model for concurrent real-time component-based systems. *Journal of science and technology Vietnam*, 49(4A):435, 2011.

[9]. V. Chieu and D. V. Hung. Timed traces and their applications in specification and verifiction of

Bản quyền thuộc Tạp chí CNTT&TT

Bản quyền thuộc Tạp chí CNTT&TT

distributed real-time systems. In *Proceedings of the Third International Symposium on Information and Communication Tecnology 2012*. IEEE Computer Society, 2012.

[10]. S. Chouali and A. Hammad. Formal verification of components assembly based on sysml and interface automata. *Innov. Syst. Softw. Eng.*, 7(4):265–274, Dec. 2011.

[11]. M. Chupilko and A. S. Kamkin. Runtime verification based on executable models: On-the-fly matching of timed traces. In *Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013.*, pages 67–81, 2013.

[12]. Dang Van and H. Truong. Modeling and specification of real-time interfaces with utp. In Z. Liu, J. Woodcock, and H. Zhu, editors, *Theories of Programming and Formal Methods*, volume 8051 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2013.

[13]. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120. Press, 2001.

[14]. Diekert and Y. MĂ©tivier. Partial commutation and traces, 1997.

[15]. Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.

[16]. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 79–88, New York, NY, USA, 2008. ACM.

[17]. D'Souza. A logical study of timed distributed automata. 2000.

[18]. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science)*. Morgan & Claypool Publishers, 2006.

[19]. M. Keller. Parallel program schemata and maximal parallelism i. fundamental results. *J. ACM*, 20(3):514–537, 1973.

[20]. Leucker. On model checking synchronised hardware circuits. In J. He and M. Sato, editors, *Proceedings of the 6th Asian Computing Science Conference (ASIAN'00)*, volume 1961 of *Lecture Notes in Computer Science*, page 182–198, Penang, Malaysia, 2000. Springer, Springer.

[21]. Li, S. Chen, L. Jian, and H. Zhang. A web services mposition model and its verification algorithm based on interface automata. In *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '11, pages 1556–1563, Washington, DC, USA, 2011. IEEE Computer Society.

[22]. Lüttgen, W. Vogler, and S. Fendrich. Richer interface automata with optimistic and pessimistic compatibility. *Acta Inf.*, 52(4-5):305–336, June 2015.

[23]. Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[24]. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for mazurkiewicz traces. *Inf. Comput.*, 179(2):230–249, 2002.

[25]. S. Thiagarajan. A trace based extension of linear time temporal logic. In *LICS*, pages 438–447. IEEE Computer Society, 1994.

[26]. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. On relational interfaces. Technical Report UCB/EECS-2009-60, EECS Department, University of California, Berkeley, May 2009.

[27]. Zhan, E. Y. Kang, and Z. Liu. Component publications and compositions. In *Proceedings of the 2nd international conference on Unifying theories of programming*, UTP'08, pages 238–257, Berlin, Heidelberg, 2010. Springer-Verlag.

[28]. Zielonka. Notes on finite asynchronous automata. *ITA*, 21(2):99–135, 1987.

AUTHOR'S BIOGRAPHY

Do Van Chieu is working at Faculty of Information Technology, HaiPhong Private University. He received the Engineer and M.Sc. degrees in software engineering from Hanoi University of Science and Technology in 2003 and 2005. His research interests include Component-based Software Design, Formal Techniques for Software Specification and Verification.