Synthesis and Refinement Check of Sequence Diagrams

Hisashi MIYAZAKI†, Tomoyuki YOKOGAWA††, Members, Sousuke AMASAKI††, Kazuma ASADA††, Nonmembers, and Yoichiro SATO††, Member

SUMMARY During a software development phase where a product is progressively elaborated, it is difficult to guarantee that the refined product retains its original behaviors. In this paper, we propose a method to detect refinement errors in UML sequence diagrams using LTSA (Labeled Transition System Analyzer). The method integrates multiple sequence diagrams using hMSC (high-level Message Sequence Charts) into a sequence diagram. Then, the method translates the diagram into FSP representation, which is the input language of LTSA. The method also supports some combined fragment operators in the UML 2.0 specification. We applied the method to some examples of refined sequence diagrams and checked the correctness of refinement. As a result, we confirmed the method can detect refinement errors in practical time.

key words: UML, sequence diagram, refinement, model checking, LTSA

1. Introduction

In the development of a large software system, the system is usually divided into multiple modules according to its expected functions. These modules are progressively elaborated and the system is finally obtained by integrating the elaborated modules. Errors in elaboration may cause implementation bugs or inconsistency in specifications, and it is crucial to guarantee that the elaborated modules and the obtained system retain its original behaviors.

A sequence diagram in UML [1] is one of the popular notations for modeling behavior of a system in terms of interactions between modules. Semantics of the sequence diagram and definition of the refinement based on the semantics are provided, for instance, [2], [3]. However, few of them can express sequence diagrams in an executable format and check the refinement correctness.

Process algebras [4] are also used for formally representing interactions between processes in a concurrent system. Process algebras provide algebraic laws to manipulate and analyze process descriptions. In addition, analysis tools for some process algebra style notations have been developed. For instance, FDR is for CSP [5], and LTSA [6] is for FSP.

Formal verification for refinement relation between sequence diagrams deserves to be studied for quality systems. Some methods have been proposed for modeling sequence diagrams with process algebra [7]–[9]. However, these methods cannot handle asynchrony of sequence diagrams. These methods represent behavior of sequence diagrams as the partial order of messages, and thus they cannot represent the order of actions sending or receiving a message.

In this paper, we propose a new method to detect refinement errors in UML sequence diagrams with model checking approach [10]. In the proposed method, behavior of sequence diagrams is described with FSP and the descriptions are integrated using high-level Message Sequence Chart (hMSC) [11]. Then, it is verified using LTSA whether refined and integrated process retains its original behaviors.

We adopted the definition in [3] as semantics and refinement for interactions of sequence diagrams. Thus, the proposed method can deal asynchrony of sequence diagrams and some combined fragment operators introduced in UML 2.0. This study will make the following contributions:

• representation for asynchrony of sequence diagrams,
• implementation of the framework provided in [3] and
• automatic verification of refinement relation.

In Sect. 2, we present the structure and semantics of a sequence diagram and the definition of refinement. Section 3 shows the syntax of FSP and Sect. 4 shows how to translate sequence diagrams into FSP notations. In Sect. 5, we demonstrate the proposed method by applying it to an example. We discuss related work in Sect. 6 and the conclusions are given in Sect. 7.

2. Refinement of Sequence Diagram

2.1 Sequence Diagram

The sequence diagram is defined in UML specification and is used to describe a sequence of interactions among objects [1]. Figure 1 shows the example of sequence diagram. The sequence diagram consists of lifelines of objects and
interactions among them. An object is drawn as a box with its name inside itself. A lifeline is drawn as a dashed line descending from a bottom edge of a box. An interaction between objects is drawn as a unidirectional arrow from one lifeline to another lifeline. An arrow is labeled by a message name. Interactions are classified as synchronous or asynchronous. A synchronous and an asynchronous interaction are drawn with a solid arrowhead and an a stick arrowhead, respectively.

The semantics of sequence diagram is based on events called event occurrences, which are emitted when a message is sent or received. The semantics is represented as labeled partial ordered set \( p = (E, L, \leq_E, \lambda) \). \( E \) is a set of event occurrences. \( L \) is a set of labels. \( \leq_E \) is a partial order relation between event occurrences. \( \lambda \) is a labeling function. \( L \) contains two labels \( m! \) and \( m? \) for each message \( m \) in the sequence diagram. \( m! \) and \( m? \) correspond to sending and receiving of \( m \), respectively. A relation \( p \models S \) holds if \( p \) represents sequence diagram \( S \).

A trace is a sequence of event occurrences obtained through extending the order of \( p \) to total. A trace \( t \) is defined as \( t = (E', L', \lambda', \leq_E') \), where \( E' = E, L' = L, \lambda' = \lambda \), and \( \leq_E' \) is a total order relation satisfying \( \leq_E \leq \leq_E \). A computation is a labeled trace, and the trace \( t \) labeled by \( \lambda \) is written as \( t^\lambda \). For example, the sequence diagram in Fig. 1 has a computation \( m1m1m2m2m2m2m2m2 \). A relation \( t^\lambda \models S \) holds if \( p \models S \) and \( t^\lambda \) is obtained from \( p \). \( t^\lambda \models S \) represents \( t^\lambda \) is executable in sequence diagram \( S \). A sequence diagram \( S \) performs one of the computations \( t^\lambda \) which holds \( t^\lambda \models S \) nondeterministically.

2.2 Combined Fragment

Some kinds of control and logical structures are represented by a combined fragment in a sequence diagram. A combined fragment is drawn as a rectangle including interactions in a sequence diagram, and the rectangle is called frame. A word in upper-left corner of the frame indicates the type of a combined fragment. A frame can be separated into several regions by dashed lines. Since interactions in a frame are considered as a sequence diagram, interactions inside a frame also have traces. Traces of a frame itself can be obtained from the traces inside the frame (or regions) depending on the type of the combined fragment. Consider that \( u \) is a trace obtained from interactions outside the frame and \( v \) is one of the traces of the frame itself. Traces of a sequence diagram with a combined fragment are obtained by merging \( u \) and \( v \) to satisfy the following condition: if event occurrences \( e_u \) in \( u \) and \( e_v \) in \( v \) have the order relation \( e_u \leq_E e_v \) on the partial ordered set \( p \), then \( e_u \) is located former (latter) than \( e_v \) in \( t \).

Our proposed method can deal with five types of combined fragments: parallel (par), alternative (alt), loop (loop), weak sequencing (seq), and strict sequencing (strict). Here we suppose a trace \( w \) is obtained from interactions inside a frame and traces \( w_1, w_2, \ldots \) are obtained from regions \( r_1, r_2, \ldots \) inside the frame, respectively.

Parallel operator (par) describes a parallel merge of interactions inside a frame. In Fig. 2 (a), the interactions \( m2m2m2m2m2? \) and \( m3m3m3? \) are interleaved. A trace of the par frame can be obtained by merging \( w_1, w_2, \ldots \).

Alternative operator (alt) describes a nondeterministic selection of interactions between regions of a frame. In Fig. 2 (b), either \( m2 \) or \( m3 \) is processed after \( m1 \). A trace of the alt frame is obtained as any one of \( w_1, w_2, \ldots \).

Loop operator (loop) describes an iteration of interactions inside a frame. In Fig. 2 (c), \( m2 \) and \( m3 \) are processed iteratively after \( m1 \), and \( m4 \) follows the iteration. A trace of the loop frame is obtained by repeating the trace \( w \).

Weak sequencing (seq) describes that event occurrences inside a frame have the same order. In the case of Fig. 2 (d), the interactions \( m2 \) and \( m3 \) can be processed interleavingly. A trace of the seq frame is obtained as \( w \) itself.

Contrastingly, strict sequencing (strict) describes that interactions inside a frame are processed along their vertical positions in the frame. In the case of Fig. 2 (e), \( m3 \) is processed after \( m2 \), and \( m4 \) also follows \( m3 \). A trace of the strict frame is obtained by restricting \( w \) to satisfy the order relation determined according to vertical positions.

2.3 hMSC

A sequence diagram is also called as basic Message Sequence Chart (bMSC). The High-level Message Sequence Chart (hMSC) describes a way of combining bMSCs by using directed graph [11]. In hMSC, a node represents a sequence diagram and an edge represents possible order of those nodes. Figure 3 shows the example of hMSC. Each node (except an initial, terminal, and connecting node) is mapped to one bMSC or one hMSC. Edges can connect all kinds of nodes. Multiple edges directed from a node represent branching operation. A connecting node joins those edges.
2.4 Definition of Refinement

Following the definition of refinement of sequence diagram [3], we define a refinement relation between sequence diagrams as follows:

**Definition:** A sequence diagram \( S' \) refines a sequence diagram \( S \), written as \( S \rightarrow S' \), if for any computation \( t' \), \( t' \models S' \) implies \( t' \models (L' \setminus L) \models S \), where \( t' \models L \) represents the computation which is obtained by excluding labels in \( L \) and \( L' \setminus L \) represents the relative complement of \( L \) in \( L' \).

This definition represents inclusion relation of the set of computations. That is, a computation executable in \( S' \) is also executable in \( S \). This also means that a set of computations cannot be enlarged by refinement.

Figure 4 shows examples of refinement. The sequence diagram in Fig. 4(a), we called \( S \) here, shows the system in which a user sends an id and a password to an interface. Sequence diagrams of Figs. 4(b) and (c), we called \( S' \) and \( S'' \), refine \( S \) so that after a user sends an id, an interface also sends the id to a server and receives an acknowledgement from the server. The user and the interface treat a password as well as the id in \( S' \) and \( S'' \). In \( S' \), the interface also sends an acknowledgement to the user after receiving the acknowledgement. Contrastingly, the interface of \( S'' \) does not send the acknowledgement to the user.

\( S \) has only one computation \( t = id!id?p!pw!pw \). The only computation in \( S' \) is the sequential processing of all messages, \( t' = id!id?snd_id!snd_id? \cdots \). \( t \) can be obtained from \( t' \) by excluding the labels of the messages which are not included in \( S' \). Therefore, a relation \( S \rightarrow S' \) holds.

In \( S'' \), the user can send the message \( pw \) before the message \( id \) is received because the interface does not send the acknowledgement to the user. Therefore \( S'' \) has a computation \( t'' = id!pw!id? \cdots \). However, \( t \) cannot be obtained from \( t'' \) and thus \( S'' \) does not refine \( S \).

3. LTSA

Labeled Transition System Analyzer (LTSA) is a tool which supports model checking of software components modeled as a set of finite state automata [6]. LTSA can check absence of deadlocks and more general properties that represent system requirements. The input language of the tool is a process algebra style notation, called Finite State Processes (FSP). The LTSA tool also features graphical display of LTSs, interactive simulation, graphical animation of behavior models, and automatic verification.

3.1 FSP

A process is defined as an ordered set of actions in FSP and is described with arrow operators \( \rightarrow \) and labels as follows:

\[ P = (\text{label}_1 \rightarrow \text{label}_2 \rightarrow \cdots \rightarrow \text{label}_m \rightarrow P). \]

For example, the following definition describes the process \( P \) which repeatedly engages in the actions \( a \) and \( b \).

\[ P = (a \rightarrow b \rightarrow P). \]

FSP predefines a local process \( \text{END} \), which engages in no further actions. The following example is a process \( Q \) which does actions \( a \) and \( b \) and then terminates.

\[ Q = (a \rightarrow b \rightarrow \text{END}). \]

The process \( P \) can also be defined as follows.

\[ P = (a \rightarrow R), \]

\[ R = (b \rightarrow P). \]

\( R \) is a local process of \( P \), and its scope is limited in the definition of \( P \).

Parallel composition of processes constructs a transition system which allows possible interleavings of actions shared by composed processes. The parallel composition is described with the \( || \) operator in FSP. In the case that processes share the same action, interleavings are restricted till those processes carry out the action. That is, shared actions synchronize executions of multiple processes. For example, parallel composition of two processes \( Q \) and \( R \) are described as follows:

\[ Q = (\text{in} \rightarrow \text{sync} \rightarrow \text{sync} \rightarrow \text{Q}). \]

\[ R = (\text{sync} \rightarrow \text{out} \rightarrow \text{sync} \rightarrow \text{R}). \]

\[ ||P = (Q \mid R). \]

The process \( Q \) repeatedly engages in the actions \text{in}, \text{sync}, and \text{sync}. The process \( R \) does \text{sync}, \text{out}, and \text{sync}. The process \( P \) is the parallel composition of \( Q \) and \( R \). \( P \) repeatedly engages in the actions \text{in}, \text{sync}, and \text{sync} because \( Q \) and \( R \) are synchronized by the shared action \text{sync}.

The choice of actions is described with the \( | \) operator.
P = (x->Q | y->R).

This example describes a process which initially engages in either of the actions x or y. If the action x is performed, subsequent behavior is described by Q, or by R if the action y is performed. Non-deterministic choice is expressed by having the same action leading to different succeeding behaviors.

Hiding operator \ makes actions in a process concealed and silent. The silent actions are labeled \(\tau\).

\[ Q = (\text{in}->\text{sync}->\text{sync}->Q) \setminus \{\text{sync}\}. \]

\[ R = (\text{sync}->\text{out}->\text{sync}->R). \]

\[ ||P = (Q || R). \]

Process Q and R share the action \text{sync} and thus they are expected to be synchronized. However, \text{sync} is hidden in the process Q and thus Q and R are not synchronized.

3.2 Safety Properties

LTSA can check safety and liveness properties as well as properties expressed in temporal logic LTL. In this paper, we carry out the refinement checking based on the safety property verification using LTSA. Informally, a property specifies a set of acceptable behavior for a system it is composed with. Safety properties of a system can be specified by using deterministic primitive processes. A safety property process is denoted by the keyword property. For example, the following property specifies behavior in which \(x\) must occur before \(y\) occurs.

property SPC = (x->y->SPC).

This property can be represented as the automaton shown in Fig. 5. If sequences of actions of a system violates this property, the process SPC is led to the error state, which is represented as -1.

The safety property is checked on a composition of a system and its property process. When sequences of actions are all acceptable, the property process does not restrict any operation of the system and accepts all possible interleavings. If the property is violated, the property process is led to an error state and the system falls into deadlock. LTSA generates an automaton which represents an action sequence to the error state.

For example, we check whether the following systems P1 and P2 satisfy the safety property SPC.

\[ P1 = (x->a->y->P1). \]

\[ P2 = (x->y->x->P2). \]

\[ ||REF1 = (SPC || P1). \]

\[ ||REF2 = (SPC || P2). \]

REF1 is the parallel composition of SPC and P1, and REF2 is the parallel composition of SPC and P2. The automata generated by LTSA is shown in Fig. 6. REF1 never reaches the error state and thus satisfies the safety property of SPC. On the other hand, REF2 reaches the error states by the sequence of actions x, y, x and x. Consequently, P2 does not satisfy the safety property and the violating sequence is generated as the automaton shown in Fig. 6.

4. Refinement Checking with LTSA

4.1 FSP Representation for Sequence Diagram

In our method, an event occurrence \(e^j\) in a sequence diagram is represented as an action in FSP notation. Here, \(e^j\) denotes the \(j\)th event occurrence on lifeline \(Obj_j\). A lifeline is represented as a process which engages in the actions corresponding to event occurrences on that lifeline. The order of actions represents the order relations between event occurrences.

If no event occurrence sends a synchronous message on \(Obj_j\), the process \(P_i\) for \(Obj_j\) can be defined as follows:

\[ P_i = (e_{i,1}->e_{i,2}->\cdots->e_{i,n}->\text{END}). \]

where, \(e_{i,j}\) denotes an action corresponding to \(e^j\). Contrastingly, in the case that \(e^j\) sends a synchronous message, the following event occurrences must be processed after the message is completely received. The event occurrence representing the receiving the message is written as \(e^j\). \(P_i\) having such an event occurrence is defined as follows.

\[ P_i = (e_{i,1}->e_{i,2}->\cdots->e_{i,j}->e_{k,1}
\]
\[ ->e_{i,j+1}->\cdots->e_{i,n}->\text{END}). \]

The order relations between sending and receiving messages can also be represented by using process description. The following process \(M_k\) represents the order relation between sending and receiving message \(m_k\).

\[ M_k = (m_k_{\text{snd}}->m_k_{\text{rcv}}->M_k). \]

Here, the action \(m_k_{\text{snd}}\) and \(m_k_{\text{rcv}}\) correspond to the event occurrences labeled as \(m_k\) and \(m_k\) respectively.

The set of computations each of which is executable in a sequence diagram is represented by using parallel composition of processes corresponding to those computations. Processes for order relations can also be composed together. Therefore, we can represent the sequence diagram \(S\) as the process \(S\) as follows:

\[ S = (P1 || \cdots || P_n || M_1 || \cdots || M_k). \]
4.2 FSP Representation for Combined Fragments

Suppose that frame $f_k$ is located between events $e^i_l$ and $e^i_{l+1}$ on the lifeline $O bj_i$ as shown in Fig. 7. Our method first defines actions $f_k.b$ and $f_k.e$ at the beginning and the end of $f_k$, respectively. The following process $P_i$ represents $O bj_i$ with $f_k.b$ and $f_k.e$:

$$P_i = (e^i_1->e^i_2->\cdots->e^i_j$$
$$->f_k.b->f_k.e->e^i_{j+1}->\cdots->e^i_n->END).$$

Our method then defines a process $F_k$ for the frame $f_k$ according to the types of combined fragments.

4.2.1 par Operator

Inside $par$ frame, interactions in distinct regions are interleaved. Thus, interactions in the same region are expressed as a process similar to a basic sequence diagram and those processes are synchronized at the beginning and the end of the regions. To synchronize the beginning and the end of the regions, all processes have the same actions $f_k.b$ and $f_k.e$ at the first and the last.

For example, consider $par$ frame $f_k$ having $n$ separated regions $r^1, r^2, \ldots, r^n$. We define a function $prefix(Obj_i, r^i)$ which returns the prefix of the process expressing the behavior of $Obj_i$ in the region $r^i$. For example, when an object $Obj$ sends a message $m$ in a region $r$, the behavior is expressed as a process $m_{snd}->END$. That is, $prefix(Obj, r)$ returns $m_{snd}$. Using this function, processes $F_k.P_i$ for objects $Obj_i$ in the region $r^i$ are defined as follows:

$$F_k.P_i = (f_k.b->prefix(Obj_i, r^i)->f_k.e->END).$$

The processes $F_k.P_i$ for objects $Obj_i$ in frame $f_k$ are defined by the parallel composition of $F_k.P_i$ for all regions $r^i$ of the frame $f_k$ as follows:

$$||F_k.P_i = (F_k.P_i.1)||\cdots||F_k.P_i.n).$$

The process $F_k$ for the frame $f_k$ is also defined by the parallel composition of $F_k.P_i$ as follows:

$$||F_k = (F_k.P1)||\cdots||F_k.PN).$$

4.2.2 alt Operator

Inside $alt$ frame $f_k$, interactions in any of distinct regions $r^1, r^2, \ldots, r^n$ are selected nondeterministically. We represent selecting a region $r^i$ as an action $alt.k.1$ and the processes $F_k.P_i$ for objects $Obj_i$ in frame $f_k$ are defined as follows.

$$F_k.P_i = (f_k.b->F_k.P_i.ALT),$$
$$F_k.P_i.ALT = (alt.k.1->prefix(Obj_i, r^i)->f_k.e->END$$
$$\cdots$$
$$|alt.k.n->prefix(Obj_i, r^n)->f_k.e->END).$$

As same as $par$ frame, the process $F_k$ is the parallel composition of $F_k.P_i$ for all objects $Obj_i$ involved in $f_k$.

4.2.3 loop Operator

Inside $loop$ frame, interactions in $f_k$ are processed iteratively. Defining a function $prefix(Obj_i, f_k)$ which returns the prefix of the process expressing the behavior of $Obj_i$ in the frame $f_k$, processes $F_k.P_i$ for $Obj_i$ in the frame $f_k$ are defined as follows:

$$F_k.P_i = (f_k.b->F_k.P_i.LBGN),$$
$$F_k.P_i.LBGN = (prefix(Obj_i, f_k)->F_k.P_i.LEND),$$
$$F_k.P_i.LEND = (f_k.e->END|loop.k->F_k.P_i.LBGN).$$

Iterations of objects in $f_k$ are synchronized by the action $loop.k$. The process $F_k$ is represented as the parallel composition of $F_k.P_i$ for all objects $Obj_i$ involved in $f_k$.

4.2.4 seq Operator

Inside the $seq$ frame, event occurrences in a lifeline are processed in the same order as a basic sequence diagram. Thus interactions in $Obj_i$ are expressed as a process similar to a basic sequence diagram and the actions $f_k.b$ and $f_k.e$ are added to the first and the last of the process. The behavior of $Obj_i$ in $f_k$ being expressed as the process $prefix(Obj_i, f_k)->END$, the processes $F_k.P_i$ are defined as follows.

$$F_k.P_i = (f_k.b->prefix(Obj_i, f_k)->f_k.e->END).$$

The process $F_k$ is defined as the parallel composition of $F_k.P_i$ for all objects $Obj_i$.

4.2.5 strict Operator

As stated above, interactions inside $strict$ frame are processed along the predefined order. Consequently, interactions in frame $f_k$ are expressed as a process which engages in the labels in the frame along the order. Suppose there exist the messages $m_1, m_2, \ldots$ in the $strict$ frame $f_k$ in order, $F_k$ is defined as follows.

$$F_k = (f_k.b->m_{1}_{snd}->m_1.rcv->m_{2}_{snd}->m_2.rcv$$
$$\cdots->f_k.e->END).$$
diagrams (bMSCs) referred from an hMSC and generate a process. Figure 8 shows the example of hMSC and bMSCs. This method first labels the initial, terminal, and connecting nodes in a hMSC as BGN, END, and CON_i, · · · , CON_M, respectively. Then the first and the last of a lifeline Ob ji of a bMSC bk is labeled as BGN_k,i and END_k,i. Behavior of Ob ji of bMSC bk is expressed as a process BGN_k,i with a local process END_k,i. A process END_k,i is also defined with a local process BGN_m,i, where bk are connected to bm in an hMSC. For example, Ob j1 in Fig. 8 is defined as the following process P1.

\[
P1 = \text{BGN}, \\text{BGN} = \text{BGN}_1, \\text{BGN}_1 = (m_1 \text{snd} \rightarrow \text{END}_1), \\text{END}_1 = (\text{BGN}_2|\text{BGN}_3), \\text{BGN}_2 = (m_2 \text{snd} \rightarrow \text{END}_2), \\text{END}_2 = \text{CON}_1, \\text{BGN}_3 = (m_3 \text{snd} \rightarrow \text{END}_3), \\text{END}_3 = \text{CON}_1, \\text{CON}_1 = \text{BGN}_1.
\]

If a process have no action then its local process can be reduced by replacing the local process with its definition. For example, the local processes BGN and BGN_1,1 in the above process can be omitted as follows.

\[
P1 = (m_1 \text{snd} \rightarrow \text{END}_1),
\]

As a result, P1 is reduced as follows.

\[
P1 = \text{BGN}_1, \\text{BGN}_1 = (m_1 \text{snd} \rightarrow \text{END}_1), \\text{END}_1 = (\text{BGN}_2|\text{BGN}_3), \\text{BGN}_2 = (m_2 \text{snd} \rightarrow \text{BGN}_1), \\text{BGN}_3 = (m_3 \text{snd} \rightarrow \text{BGN}_1).
\]

The integrated process SYS is then defined as follows.

\[
||\text{SYS} = (P1||P2||M1||M2||M3).
\]

4.4 Refinement Checking

We define \( S \sim S' \) if all computations of \( S' \) are also the computation of \( S \). Our definition of refinement is equivalent to trace inclusion of LTSS and it is known that weak simulation implies trace inclusion [4]. LTSA can check the property expressed as a process Q, we can obtain the following binary relation \( R \) over states of LTSS P and Q, which correspond to processes P and Q; (1) for the initial states \( p_0 \) and \( q_0 \) of \( P \) and Q, \( (p_0, q_0) \in R \) (2) if \( (p, q) \in R \) and \( p \) can reach to \( p' \) by some action \( a \) then \( (p', q') \in R \) where \( p, p' \) and \( q, q' \) are states of \( P \) and Q, and \( q \) reaches to \( q' \) by a sequence of actions \( (\tau a \tau' a) \). Such a relation \( R \) is a weak simulation [12]. Therefore refinement relation can be checked by carrying out the verification of the safety property where \( S' \) is the system process and \( S \) is the property process. The system process SYS, the property process SPC, and the parallel composition of them are defined as follows.

\[
\begin{align*}
\text{property} \ & ||\text{SPC} = (P1||\cdots). \\
\text{SYS} \ & ||\text{SYS} = (P1||\cdots)||\text{m_snd, \cdots}. \\
\text{REF} \ & ||\text{REF} = (\text{SYS}||\text{SPC}).
\end{align*}
\]

The actions in the system process but not in property process SPC are concealed. If there exist behaviors of SYS which cannot satisfy the behaviors of SPC, REF reaches to the error state and LTSA detects refinement error as the deadlock. With this modeling, LTSA can also specify an event occurrence which causes refinement violation as the action on the transition up to the error state.

5. An Example

Figure 9 shows a sequence diagram and its refinements. The sequence diagram in Fig. 9(a) has strict frame. The sequence diagram in Fig. 9(b) refines it so that lifeline Obj2 interacts with a new object Obj3 through two interactions m4 and m5. The sequence diagram in Fig. 9(c) has the same interactions and lifelines as those in Fig. 9(b). However, seq frame are replaced by strict frame. Therefore the sequence diagram in Fig. 9(c) can perform the computation \( \ldots m2!m3!m2!m3!\cdots \), which the original diagram cannot. This causes the refinement violation between Figs. 9(a) and (c). We demonstrate that our method can detect this refinement violation by LTSA. We experimented on a 32-bit Windows XP personal computer with 3 GHz Intel Xeon CPU and 3.25 GB RAM.

The sequence diagram before refinement is defined as the following process SPC.

\[
P1 = (m_1 \text{snd} \rightarrow m_1 \text{rcv} \rightarrow f1_b \rightarrow f1_e \rightarrow \text{END}). \\
P2 = (m_1 \text{rcv} \rightarrow f1_b \rightarrow f1_e \rightarrow \text{END}). \\
F1 = (f1_b \rightarrow m_2 \text{snd} \rightarrow m_2 \text{rcv} \rightarrow m_3 \text{snd} \rightarrow m_3 \text{rcv} \rightarrow f1_e \rightarrow \text{END}).
\]
Fig. 9 Examples of refinements.

M1 = (m1_snd -> m1_rcv -> M1).
M2 = (m2_snd -> m2_rcv -> M2).
M3 = (m3_snd -> m3_rcv -> M3).

\[
\text{property} \mid \text{SPC} = (P1 \mid P2 \mid M1 \mid M2 \mid M3 \mid F1).
\]

SPC is defined as the property process.

Then the sequence diagram in Fig. 9(b) is defined as the following process SYS1:

R1P1 = (m1_snd -> m1_rcv -> f1b -> f1e -> END).
R1P2 = (m1_rcv -> m4_snd -> m4_rcv -> m5_rcv -> f1b -> f1e -> END).
R1P3 = (m4_rcv -> m5_snd -> END).
R1F1 = (f1b -> m2_snd -> m2_rcv -> m3_rcv -> f1e -> END).
M4 = (m4_snd -> m4_rcv -> M4).
M5 = (m5_snd -> m5_rcv -> M5).

\[
\mid \mid \text{SYS1} = (R1P1 \mid R1P2 \mid R1P3 \\
\mid M1 \mid M2 \mid M3 \mid M4 \mid M5 \\
\mid R1F1 \\
\backslash \{m4_snd,m4_rcv,m5_snd,m5_rcv\}).
\]

The sequence diagram in Fig. 9(c) is similarly defined as the process SYS2:

R2P1 = (m1_snd -> m1_rcv -> f1b -> f1e -> END).
R2P2 = (m1_rcv -> m4_snd -> m4_rcv -> m5_rcv -> f1b -> f1e -> END).
R2P3 = (m4_rcv -> m5_snd -> END).
R2F1P1 = (f1b -> m2_rcv -> m3_rcv -> f1e -> END).
R2F1P2 = (f1b -> m2_snd -> m3_snd -> f1e -> END).

\[
\mid \mid \text{SYS2} = (R2P1 \mid R2P2 \mid R2P3 \\
\mid M1 \mid M2 \mid M3 \mid M4 \mid M5 \\
\mid R2F1P1 \mid R2F1P2 \\
\backslash \{m4_snd,m4_rcv,m5_snd,m5_rcv\}).
\]

Refinement are checked on composed process of the system processes SYS1 and SYS2, and the property process SPC:

\[
\mid \mid \text{REF1} = (\text{SPC} \mid \mid \text{SYS1}).
\]

Figures 10 and 11 shows the automata generated by LTSA corresponding to REF1 and REF2, respectively. The time required for composition of REF1 is 186 ms and 4514 KBytes memory are used. On the other hand, the time required for composition of REF2 is 203 ms and 4506 KBytes memory are used.

Figure 10 indicates that REF1 does not reaches the error state. This result implies that the sequence diagram in Fig. 9(b) correctly refines the sequence diagram in Fig. 9(a). In contrast, Fig. 11 indicates that REF2 falls into the error state if the action m3_snd is performed at the 4th state. Thus the refinement violation can be detected successfully and it turns out that the sequence diagram in Fig. 9(c) has unintended behavior, that is, sending m3.

6. Related Work

Existing methods dealing with the sequence diagrams focused on various aspects of them: timing consistency [13], [14], real-time properties [15], consistency with other diagrams [7], [16], [17] and so on. Several methods for defining the semantics of sequence diagrams have been proposed, and a number of methods for modeling sequence diagrams have also been developed. In recent studies [18], [19], Modal Transition Systems (MTS) are used to define refinement relation of sequence diagrams. MTS is an extension of LTS and represents behavior as a set of states and a set of required or potential transitions. As stated above, the definition on [3] we adopted is equivalent to the weak simulation on LTSs. Thus our method can be implemented by any environments based on LTSs without expressing modal behavior.

Modeling sequence diagrams with process algebra has also been studied [7]–[9], [20], [21]. Korenblat et al. [7] proposed the method for modeling UML state machine diagrams and sequence diagrams by π-calculus. This method provided the coherence of the two types of diagrams and a formal semantics for them.

Tribastone et al. [8] proposed the method to carry out quantitative evaluation of sequence diagrams. This method used the stochastic process algebra PEPA [22] as a performance engine for MARTE [23]. They also presented an automatic translation from sequence diagrams to PEPA process algebra models.
Uchitel et al. [9], [20] presented the procedure to integrate multiple sequence diagrams into an FSP and carry out formal verification using LTSA. This framework for the integration is based on the use of hMSCs. In addition, a workbench for supporting existing integration approaches was provided.

On the other hand, Mitchell [21] defined process algebra which is an extension of the Mauw and Reniers’ algebraic semantics [24] for sequence diagrams. The proposed process algebra supports the causal semantics [25].

These approaches represent the behaviors of sequence diagrams with process algebra as the partial order of interactions, not as the partial order of event occurrences. To express the behaviors of sequence diagrams in terms of interactions can make the modeling step more simple. In addition, adopting abstract semantics may reduce the number of states and cost for verifying systems. In such semantics, however, synchronous and asynchronous messages cannot be distinguished and thus asynchronous behaviors of objects cannot be expressed. As an example, the semantics in terms of partial order of event occurrences needs to be defined before a transition of the automata. In Bernardi’s method, total transition model of statecharts and thus dealing with combination of synchronous and asynchronous messages is not allowed. Asynchrony of the sequence diagrams is an important feature for development of distributed system. In addition, to check the refinement relation between sequence diagrams, the behaviors of the diagrams need to be modeled elaborately.

Other approaches to model asynchrony of a sequence diagram are also presented [16], [17], [26], [27]. Krüger et al. [26] proposed the method to generate statecharts from a set of Message Sequence Charts [28]. Ziadi et al. [27] proposed to generate statecharts from sequence diagrams and defined algebraic framework for composing statecharts. These methods represent asynchrony of sequence diagrams clearly by the parallel composition of statecharts. However, synthesis of state machines depends on communication model of statecharts and thus dealing with combination of synchronous and asynchronous messages is not allowed.

Zhao et al. [16] proposed the method for verifying the consistency between sequence diagrams and state machine diagrams using SPIN [29]. The sequence diagrams are expressed as automata. Bernardi et al. [17] provided the translation from sequence diagrams and state machine diagrams into Generalized Stochastic Petri Nets. However, Zhao’s method expressed the interaction of sequence diagrams as a transition of the automata. In Bernardi’s method, total order to all event occurrences needs to be defined before translation to petri nets. Thus all computations of sequence diagrams cannot be represented in those methods.

7. Conclusion

In this paper, we proposed the method to integrate sequence diagrams using hMSC and verify the refinement relation between them. To this purpose, we first presented the translation from sequence diagrams to FSP, which is the input language of LTSA. We also show FSP representation of behaviors describe by five types of combined fragments. Our method can integrate the sequence diagrams using hMSC. As a case study, we applied the proposed translation to sample diagrams and showed our method can detect the refinement violation.

We considered only the valid computation of a sequence diagram. It is necessary to introduce the concept of the invalid computation [2] to extend the method to neg or assert operators.

References

[1] Object Management Group, Unified Modeling Language Specification, Object Management Group, 2001. http://www.uml.org
[2] H. Störrle, “assert, negate and refinement in UML-2 interactions,” Proc. 2nd Int’l Workshop on Critical Systems Development with UML (CSDUML’03), pp.79–93, 2003.
[3] M.V. Cengarle and A. Knapp, “UML 2.0 interactions: Semantics and refinement,” Proc. 3rd Int’l Workshop on Critical Systems Development with UML (CSDUML’04), pp.85–99, 2004.
[4] R. Milner, A Calculus of Communicating Systems, Springer-Verlag New York, Secaucus, NJ, USA, 1982.
[5] C.A.R. Hoare, “Communicating sequential processes,” Commun. ACM, vol.21, pp.666–677, Aug. 1978.
[6] J. Magee and J. Kramer, Concurrency: State Models & Java Programs, John Wiley & Sons, New York, NY, USA, 1999.
[7] K. Pokozy-Korenblat and C. Priami, “Toward extracting $\pi$-calculus from UML sequence and state diagrams,” Electron. Notes Theor. Comput. Sci., vol.101, pp.51–72, 2004.
[8] M. Tribastone and S. Gilmore, “Automatic translation of UML sequence diagrams into PEPA models,” Proc. 5th Int’l Conf. on Quantitative Evaluation of Systems (QEST’08), pp.205–214, 2008.
[9] S. Uchitel and I. Kramer, “A workbench for synthesising behaviour models from scenarios,” Proc. 23rd Intl. Conf. on Software Engineering (ICSE’01), pp.188–197, 2001.
[10] E. Clarke, O. Grumberg, and D. Peled, Model Checking, MIT Press, 1999.
[11] R. Alur and M. Yannakakis, “Model checking of message sequence charts,” Proc. 10th Int’l Conf. on Concurrency Theory (CONCUR’99), pp.114–129, 1999.
[12] R. Milner, Communicating and Mobile Systems: The $\pi$-calculus, Cambridge University Press, New York, NY, USA, 1999.
[13] X. Li and J. Lilius, “Timing analysis of UML sequence diagrams,” Proc. 2nd Int’l Conf. of the Unified Modeling Language (UML’99), pp.661–674, 1999.
[14] X. Li and J. Lilius, “Checking compositions of UML sequence diagrams for timing inconsistency,” Proc. 7th Asia-Pacific Software Engineering Conf. (APSEC’00), pp.154–161, 2000.
[15] K. Lano, “Formal specification using interaction diagrams,” Proc. 5th IEEE Int’l Conf. on Software Engineering and Formal Methods (SEFM’07), pp.293–304, 2007.
[16] X. Zhao, Q. Long, and Z. Qiu, “Model checking dynamic UML consistency,” Proc. 8th Int’l Conf. on Formal Engineering Methods (ICFEM’06), pp.440–459, 2006.
[17] S. Bernardi, S. Donatelli, and J. Merseguer, “From UML sequence diagrams and statecharts to analysable petri net models,” Proc. 3rd
Int’l Workshop on Software and Performance (WOSP’02), pp.35–45, 2002.

[18] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic, “Synthesizing partial component-level behavior models from system specifications,” Proc. 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE’09), pp.305–314, 2009.

[19] S. Uchitel, G. Brunet, and M. Chechik, “Synthesis of partial behavior models from properties and scenarios,” IEEE Trans. Softw. Eng., vol.35, no.3, pp.384–406, May 2009.

[20] S. Uchitel, J. Kramer, and J. Magee, “Synthesis of behavioral models from scenarios,” IEEE Trans. Softw. Eng., vol.29, no.2, pp.99–115, 2003.

[21] B. Mitchell, “Characterizing communication channel deadlocks in sequence diagrams,” IEEE Trans. Softw. Eng., vol.34, no.3, pp.305–320, 2008.

[22] J. Hillston, A Compositional Approach to Performance Modelling (Distinguished Dissertations in Computer Science), Cambridge University Press, New York, NY, USA, 2005.

[23] Object Management Group, UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE).

[24] S. Mauw and M. Reniers, “An algebraic semantics of basic message sequence charts,” Comp. J., vol.37, pp.269–277, 1994.

[25] C. Sibertin-Blanc, O. Tahir, and J. Cardoso, “Interpretation of UML sequence diagrams as causality flows,” Proc. 5th Int’l School and Symp. on Advanced Distributed Systems (ISSADS’05), pp.126–140, 2005.

[26] I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to statecharts,” Proc. IFIP WG10.3/WG10.5 Int’l Workshop on Distributed and Parallel Embedded Systems (DIPES’98), pp.61–71, 1999.

[27] T. Ziadi, L. Helouet, and J.M. Jezequel, “Revisiting statechart synthesis with an algebraic approach,” Proc. 26th Int’l Conf. on Software Engineering (ICSE’04), pp.242–251, 2004.

[28] Message Sequence Charts (MSC), ITU-T. Z.120 ed., 1999.

[29] G.J. Holzmann, “The model checker SPIN,” IEEE Trans. Softw. Eng., vol.23, no.5, pp.279–295, 1997.

Hisashi Miyazaki received the M.E. degree from Okayama Prefectural University in 2005. He is currently an assistant professor in Kawasaki University of Medical Welfare and studying towards the Ph.D. degree in Okayama Prefectural University. He has been engaged in research on automatic verification of formal specifications.

Tomoyuki Yokogawa received the M.E. and Ph.D. degrees from Osaka University in 2000 and 2003. He is currently an assistant professor in Okayama Prefectural University, Japan. His research interests include formal verification and software modelling.

Sousuke Amasaki received the Ph.D. degree in information science and technology from Osaka University, in 2006. He is currently an assistant professor in Okayama Prefectural University, Japan. His research interests include empirical software engineering, software prediction systems, and software quality assurance.

Kazuma Asada is currently studying towards the M.E. degree in the Graduate School of Systems Engineering at Okayama Prefectural University. He has been engaged in research on model checking for UML sequence diagrams.

Yoichiro Sato received the M.E. degree in electronic engineering from Okayama University in 1982 and Ph.D. degree in electronic engineering from Osaka University in 1990. He is currently an associate professor in Okayama Prefectural University, Japan. His research interests include asynchronous circuit design.