A Component-Based Approach for Consistency Checking between
UML State and Sequence Diagrams

Youcef HAMMAL
LSI, Département d’Informatique, Faculté d’Electronique & Informatique
Université des Sciences et de la Technologie Houari Boumediene
BP 32, El-Alia 16111, Bab-Ezzouar, Algiers, Algeria.
E-MAIL : yhammal@wissal.dz

Abstract. This paper presents a component-based methodology for consistency checking between state diagrams and the relating sequence diagrams. In order to avoid the state explosion problem, first a partial reachability graph is generated from the set of state diagrams with regards to some component. Likewise, we extract a partial behavioral graph from the set of available sequence diagrams wherever this component is involved as participant. Afterwards, we check whether the former graph simulates the latter graph modulo some preorder relation. The main advantage of this approach is to allow us to achieve a modular checking of consistency between above diagrams. It copes as well with the handling of adaptability and substitutability checking of any component at an early stage of the process.

1. Introduction

The Unified Modeling language (UML [15]) is a multi-paradigm language which copes well with modeling and design of complex systems by offering many visual and flexible notations for description of their various aspects. Also, UML includes high-quality design concepts such as abstraction, encapsulation and decomposition of systems into components.

Note that complex systems are typically large and reactive systems including components that may operate concurrently by means of various synchronization and communication mechanisms. Therefore, modeling and analysis of concepts like reliability and safety become as important as functional requirements.

Although UML is an OMG standard language [15] provided with precise syntactic aspects for its notations and supported by wide-established design tools, their semantics remain too imprecise and lack verifiability capabilities [4,6,8,11,20,22].

So considering the semi-formal aspect of UML, many methods are proposed to embed UML diagrams into formal modeling languages to take advantage of their tools developed for analysis, simulation and verification of parts of produced models.

Likewise, our approach focuses on ascribing behavioral diagrams of UML (mainly Sequence and State Diagrams) with formal semantics in terms of mathematical denotations in order to allow both consistency verification of the different dynamic diagrams and analysis of timeliness and performance of an UML model.

In fact, software designers use interaction diagrams particularly sequence diagrams to depict how system components and users interact in order to provide system level functionality. They are also used during the more detailed design phase where the precise inter-process communication must be specified according to formal protocols [15]. On the other hand side, the behavior of each component involved in these interactions, is described by means of a high level state machine called state diagram.

However in spite of the expressiveness and precise syntactic aspects of the two above UML notations, their semantics remain described in natural language with sometimes OCL formulas. Hence no formal way is available to verify whether the interactions among components of some concurrent system under design are consistently realizable by the set of StateCharts depicting intra-components behaviors.

Accordingly, to allow using automated tools for analysis, simulation and verification of UML diagrams, we should assign them precise and formal semantics by means of rigorous mathematical formalisms [6,13,4,19,20].
In this context, we proposed in a previous work [6] a new translation method from State Diagrams into timed Petri nets. Once the entire StateChart of the system is translated into a timed Petri net, one could achieve a consistency validation of the dynamic view with respect to behavioral models of sequence diagrams.

We defined also branching time semantics for UML Sequences Diagrams [7] yielding behavioral graphs that specify faithfully the intended behaviors by recording both traces of all interaction components together with branching.

In this paper, we present a component-based methodology for consistency checking between state diagrams and relating sequence diagrams (see fig.1). In order to avoid the state explosion, first we give a method to generate a partial reachability graph from the Petri net of StateCharts with regards to some component. Likewise, we extract a partial behavioral graph from the set of available sequence diagrams where this component is involved as participant. Afterwards, we present some preorder relations to check whether our reachability graph can be simulated by the second graph.

The main benefit of this approach is to allow us achieving a modular checking of consistency between above diagrams. It copes as well with the handling of substitutability checking of any component at early stages of development process.

The remainder of the paper is organized as follows: Section 2 presents motivations of the component-based approach for consistency checking between UML Diagrams. Then section 3 summarizes the formalization method of state diagrams by Petri nets and section 4 gives algorithms producing partial reachability graphs. Next we present in section 5 the mapping from sequence diagrams into partial behavioral graphs. Then we present in section 6 some preorder relation among previous graphs. Finally a conclusion is given in section 7.

2. Motivation and Related Work

Nowadays design and implementation of great systems become increasingly based on distributed and interoperable components each of which encapsulates one or more services so that the composite systems yield news services to the environment. Moreover software systems evolve throughout the product life cycle; software modules are transformed as requirements change, bugs are discovered and fixed [2]. Evolution implies the removal of a previous component and addition of a new one.

In UML settings, requirements elicitation is achieved through sequence diagrams showing interactions between components and then their individual realizations are given in terms of communicating StateCharts.

Since UML StateCharts lack formal semantics, we use the formalization method defined in [6] to translate them into a rigorous formalism, namely interval time Petri nets, modeling the intended behavior of the whole system. Afterward, the reachability graph can be built where arcs are labeled with events.

This exhaustive exploration of the reachability graph may be performed by means of any classical enumeration algorithm. But construction and exploration of such graphs for complex systems mostly raise a state explosion. Furthermore we frequently need just to check whether a new component can be substituted to another one without introducing bugs like deadlock or new unsuitable behaviors.
So we propose herein a method of partial exploration of the reachability graph with respect to some main component such that we generate only partial paths of this main component interleaved to those of other components necessary to unlock execution of the main one.

Note that generation of graphs more compact than the whole state space is a practical mean often used to avoid the common state explosion problem. Hence besides works on checking on the fly [9], various approaches were addressed to explore some parts of reachability graphs so that some properties could be checked or preserved such as partial order based techniques [5,14,16,18,1].

Similarly, our component-based method can be used for consistencies checking of components like testing the lack of deadlocks in partial graphs of new components or checking that all (or parts of) sequences of events in a partial graph related to a previous component remain in the partial graph related to a new component.

Furthermore this method allows verifying in a modular way that sequence diagrams are realizable by StateCharts. Likewise generating partial reachability graph w.r.t. some component, the approach consists in extracting the component's part of the behavioral model of the sequence diagram. Thus we compare the two partial graphs by establishing some adequate preorder relation between them. Note that we discard equivalence relations since mostly sequence diagrams depict only partial views of intended behaviors of the system. In contrast to our approach supporting modular checking and based on preorder relations, authors of [12] use weak bisimulation to compare \( \pi \)-expressions depicting whole dynamic diagrams. Likewise in [22] entire class diagrams, sequence diagrams and StateCharts are transformed into description logic for checking consistency between different versions of these diagrams. Also the approach [3] verifies the consistency of sequence diagrams and StateCharts using dynamic meta-modeling (DMM) rules.

3. Mapping State Diagrams into Petri Nets

StateCharts constitute a visual formalism suitable for specification of complex and reactive systems both in terms of how components communicate and collaborate and how they carry out their own internal behavior. StateCharts describe states and transitions in a modular fashion, enabling clustering, orthogonality (i.e., concurrency) and refinement, and supporting capability for moving between levels of abstraction. Accurately, the main concepts of StateCharts are the extension of conventional state diagrams by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components [15].

Despite this language is highly diagrammatic, its semantics is not completely formal and needs improvements to provide a concrete support for inspecting and checking important properties such as checking whether the state diagram fulfills requirements depicted by sequence diagrams or whether a new component still generates the same behaviors offered by the old one or causes some deadlock situations once plugged into the evolved system.

Although many works for formalizing StateCharts have been proposed, most of them are limited to flattened state machines and ignore main concepts of StateCharts such as orthogonality, hierarchy, historicity and boundary crossing arcs. So to take advantage of StateCharts expressivity, we introduced a method of translation [6] which overcame these shortcomings by converting in a compositional manner the StateCharts into an equivalent high level Petri net modeling both event and control flows through components.

Each StateCharts related to some component is mapped into a Petri net. Then all subnets are joined together by a parallel operator to yield the whole behavior model.

We distinguish in the generated Petri net two kinds of places (see fig.2). The first one includes trigger event places supporting trigger event flows between different parts of the whole system, i.e. once some net transition fires, its related action raises an event which will trigger another action in a second step of the net execution. Moreover, a transition firing depicts also the passage from a state configuration to a new one by moving tokens throughout another kind of places. These places supporting control flow are called control events places.

The target model is an interval timed Petri net denoted by ITPN [6]. A marked ITPN is a tuple: \( R = (P, T, \text{Pre}, \text{Post}, L_1, L_2, \text{Prior}, M_I) \) where:
- \( P \) is the set of places and \( T \) is the set of transitions.
- \( \text{Pre}: T \rightarrow 2^P \) is a backward incidence function giving inplaces (input places) of transitions (\( 2^P \) denotes the powerset of \( P \)).
- Post: $T \rightarrow 2^P \times \text{INT}$ is a forward incidence function giving outplaces (output places) of transitions with the corresponding delay intervals.
- INT = \{ $<t_1,t_2> \in \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0} / t_1 \leq t_2$ \} the set of time intervals.
- $L_1$ and $L_2$ are labeling functions:
  - $L_1: P \rightarrow \Sigma$ (set of events) and $L_2: T \rightarrow \text{Act}$ (set of actions).
- Prior $\subseteq T \times T$ is a priority relationship between transitions in conflict.
- $M_0 : P \rightarrow \mathbb{N}$ is the initial marking at instant 0 from system starting.

A transition “t” is enabled within a marking $M$ if the required tokens in inplaces are available. When $t$ fires these inplaces tokens are consumed and some tokens are generated into its outplaces so that a new marking yields as follows: $M' = M \cup \text{Post}(t) \setminus \text{Pre}(t)$.

The timing policy of ITPN compels any transition to be fired as soon as it is enabled. However in this paper temporal aspects are not considered when generating the reachability graph.

### 4. Generation of the Partial Reachability Graph

#### 4.1 Algorithms

The generation of the partial graph related to some main component $C_0$ belonging to a system $S$ is based on favoring transitions of this component $C_0$.

At beginning we should execute only enabled transitions from $C_0$. Every transition, which does not need any trigger event from its environment, will fire directly. The transitions outgoing from active configurations but which require incoming events from other components will be handled differently. For instance if there is a transition $t_0$ in $C_0$ requiring a trigger event from $C_1$, we should render this second component prioritized but we restrict its execution to only sequences of transitions in $C_1$ helping $t_0$ becomes enabled. However if $C_1$ progress itself is interrupted at any transition $t_1$ which needs an incoming event from a third component $C_2$, we must perform all transitions from this component making $t_1$ enabled and so on.

This process is repeated recurrently in order to enable and perform all transitions of $C_0$ and any transitions in other components enabling those of the prior component $C_0$. Obviously to avoid infinite recurrence we should verify that any new blocked transition does not belong yet to the set of transitions we are trying to enable.

The algorithm uses a stack to store markings to explore from the initial marking. The first action of the iterative block is to pop a marking $M$. Then we compute (potentially) enabled transitions from this marking $M$ which belong to the prioritized component $C_0$. We distinguish two kinds of transitions: the first set includes enabled transitions we fire directly whereas the second set contains transitions of which control events inplaces are marked but not those supporting event flows. Intuitively such transitions denote in the StateChart an arc outgoing from an active configuration but whose trigger event is not yet available.

So to make these transitions firable, we should activate other components whose some transitions can generate the required triggering events. Note that after firing any transition, the obtained marking is pushed in the stack so that it can be handled later.

To render firable a transition “t” in the current prioritized component, we compute the set $X$ of all transitions in other components which firings produce the needed tokens in events flow inplaces of “t” so making it enabled.

Next, we handle each pair $(t',C')$ from $X$ by calling a recursive function “Unlock” which parameters are items of this pair together with the current marking and the set of transitions we want to render firable.
Algorithm Generate-Partial-Graph

Input Marking \( M_0 \), Component \( C_0 \)

Output a partial Reachability graph

Explored := \{M_0\};
Push (M_0);
while not empty (stack) do
begin
M := Pop();
Set_1 := Firable-Transitions \( (M, C) \);
for each \( t \) \in Set_1 do
begin
\( M' := \text{Successive} (M, t) \);
if \( M' \notin \text{Explored} \) then
begin
Push (M');
Explored := Explored \cup \{M'\}
end if
end for
Set_2 := Potentially-Enabled-Transitions \( (M, C) \);
for each \( t \) \in Set_2 do
Begin
Compute the set \( X \) of transitions in other components, which enable \( t \) by generating its trigger event.
for each \( (t', C') \) \in \( X \) do
begin
for each \( M' \in \text{Unlock} (M, \{t\}, t', C') \) do
begin
\( M'' := \text{Successive} (M', t) \);
if \( M'' \notin \text{Explored} \) then
begin
Push (M'');
Explored := Explored \cup \{M''\}
end if
end for
end for
end while
end Generate-Partial-Graph.

The function "Unlock" fires the transition \( t' \) if it is enabled in its component \( C' \), else it checks whether control flow inplaces of \( t' \) are marked. When these places are marked, the same dealing with \( t' \) is achieved as above by searching all transitions in other components whose firings generate event tokens making it possible to enable \( t' \).

But if control flow inplaces are not marked, we should search the sequence of previous transitions of \( t' \) to fire in the same component \( C' \) so that control flow inplaces of \( t' \) become marked (i.e. source configuration of \( t' \) becomes active) and then we resume handling of event inplaces.

At last the called function "Unlock" returns a set \( SM \) of yielded markings making it possible to enable the locked transition \( t \) in the calling routine. Note that tokens in control flow inplaces of \( t \) can not be consumed during Unlock execution because they can be used only by transitions outgoing from the same active configuration.

Function Unlock (Marking \( M \), Set of Transition \( T \), Transition \( t' \), Component \( C' \)) returns set of Marking

Set of Marking \( SM := \emptyset \);
if \( t' \) is firable from \( M \)
then Handle-Successive (M)
else
if the control flow inplace \( p \) of \( t' \) is marked then
begin
Calculate \( X \) the set of transitions which enable \( t' \) by generating a token in its trigger event inplace;
for each \( (t'', C'') \in X \) and \( t'' \notin T \cup \{t'\} \) do
begin
for each \( M' \in \text{Unlock} (M, T \cup \{t'\}, t'', C'') \) do
Handle-Successive (M');
end for
end for
end if
The subroutine Handle-Successive generates the successive marking by firing the unlocked transition \( t' \) from the marking \( M_1 \). The new marking is pushed into the stack if it has not yet explored. Then it is added to the set \( SM \) of yielded markings.

**Function** Handle-Successive(\( M_1 \))

\[
M_2 := \text{Successive}(M_1, t');
\]

if \( M_2 \) \( \notin \) Explored then

\[
\text{Push}(M_2);
\]

Exploded := Exploded \( \cup \) \{\( M_2 \)\};

end if

\[
SM := SM \cup \{M_2\};
\]

End Handle-Successive;

### 4.2 Synthesizing the whole reachability graph

Now we prove the correctness of the method by showing that the reachability graph of the whole system can be rebuilt up from the partial graphs we generate by above algorithms.

Formally, the reachability graph of a system \( S \) (denoted \( RG(S) \)) is a labeled transition system \( G=\langle Q, \Sigma, \Delta, q_0 \rangle \) where:

- \( Q \) is the set of reached markings which depict states of the system \( S \).
- \( \Sigma \) is the set of events which are labels of the net transitions.
- \( \Delta \) is the set of transitions of which firings change the markings.
- \( q_0 \) is the initial marking \( M_0 \).

Below we show how the reachability graph \( RG(S) \) can be synthesized from partial reachability graphs related to its components \( C_i \) (denoted \( RG(C_i), i \in I \)).

**Definition 1. Synchronization product of partial reachability graphs.**

Let \( G_1, G_2 \) be two graphs where \( G_1 = \langle Q^1, \Sigma^1, \rightarrow^1, q_{01} \rangle \), \( G_2 = \langle Q^2, \Sigma^2, \rightarrow^2, q_{02} \rangle \).

Synchronization product of \( G_1 \) and \( G_2 \) yields a new graph \( G_1 \otimes G_2 = \langle Q, \Sigma, \Delta, q_0 \rangle \) with:

- \( Q \subseteq Q^1 \times Q^2 \)
- \( \Sigma = \Sigma^1 \cup \Sigma^2 \)
- \( \rightarrow = \{(q, \Sigma^1) \rightarrow (q', \Sigma^2) \} / \text{if } a \in \Sigma^1 \cap \Sigma^2 \text{ then } (q_a \rightarrow q'_a) \in \rightarrow^1 \text{ and } (q_{a\rightarrow q'_a} \in \rightarrow^2) \text{ else } (q_a \rightarrow q'_a) \in \rightarrow^1 \text{ or } (q_a \rightarrow q'_a) \in \rightarrow^2 \}
- \( q_0 = (q_{01}, q_{02}) \) where \( q_{01} \) and \( q_{02} \) depict the same initial marking of the whole system.

Herein we let each graph progress if it executes some action which does not belong to the set of common events elsewise the two graphs should synchronize (see fig.3).
**Property 1.** The operation $\otimes$ over above graphs is commutative and associative.

Hence, the whole reachability graph can be built gradually by combining its partial graphs by pairs.

Note that in the partial graph related to $C_0$ there is a path $(x!; b!)$ that rises a deadlock situation. Such paths can be discovered throughout the comparison of this partial graph to sequence diagrams related to the involved component as defined in section 6.

**Lemma 1.** Let $\{C_1, \ldots, C_n\}$ be the set of components of a system $S$. The product graph $(RG(C_1)\otimes\ldots\otimes RG(C_n))$ is isomorphic to the reachability graph $RG(S)$.

Proof. Let $G_1$ be the graph $(RG(C_1)\otimes\ldots\otimes RG(C_n))$ and $G_2$ be the reachability graph $RG(S)$: $G_1 = <Q^1, \Sigma^1, \rightarrow^1, q_0^1>$, $G_2 = <Q^2, \Sigma^2, \rightarrow^2, q_0^2>$.

We should exhibit two bijections $f$ and $g$ such that: $f$: $Q^1\rightarrow Q^2$, $g$: $(\rightarrow^1)\rightarrow(\rightarrow^2)$ and $\forall t\in\rightarrow^1$ with $t=(q_1^1,a,q_2^1)$ we have $g(t) = (f(q_1^1),a, f(q_2^1))$.

We built $f$ and $g$ by induction on the length of the sequence $(t_1\ldots t_n)$.

- $f(q_0^1)=q_0^2$ because all component's graphs begin from the same initial marking $q_0^0$.
- $\forall t\in\rightarrow^1$, $t=(q_1^1,a,q_2^1)$. This means that $t$ is firable from at least one component $C_i$ in $q_0^1$, such that $q_1^i=(\ldots s_i\ldots)$ and $q_1^i\rightarrow s_i'\in\rightarrow_i$ (set of transitions of $C_i$). So following the definition 1 of "$\rightarrow^1$" there is also a transition $t$ from $q_0^1$ which depicts also the same initial marking as $q_1^1$ such that: $q_0^1\rightarrow s_i\rightarrow q_1^i$ and $f(q_1^i)=q_2^1$.
- Vice versa, each transition from $q_0^1$ should fire from one component $C_i$.
- Suppose this property is true up to some sequence $(t_1\ldots t_n)$, so we have $f(q_1^i)=q_2^i$.

![Fig.3. Partial reachability graphs related respectively to the component $C_0$, $C_1$ and $C_2$.](image-url)
- $\forall t \in \rightarrow^1, t = (q^1_n, a, q^1_{n+1})$. This means that $t$ is firable from at least one component $C_i$ in $q^1_n$, such that $q^1_n = (s_i,...,s_i)$ and $q^1_{n+1} = (s'_i,...)$ and $s_i \rightarrow a \rightarrow s'_i \in \rightarrow_i$.
- As $f(q^1_n) = q^2_n$, there should be a sequence of firable transitions $(t_0...t_n)$ such that $q^2_0 \rightarrow (t_0...t_n) \rightarrow q^2_n$.

Additionally $q_i$ is a part of $q^2_n$. Following the definition 1 of "\rightarrow" this means that transition $t$ is also firable from $q^2_n$ such that $q^2_n \rightarrow a \rightarrow q^2_{n+1}$ and the obtained state $q^2_{n+1}$ models the same marking $q^1_{n+1}$ because this one is obtained from $q^1_n$ which is identical to $q^2_n$. So $q^2_{n+1}$ has exactly the same outgoing transitions as $q^1_{n+1}$. Hence, $f(q^1_{n+1}) = q^2_{n+1}$.

5. Sequence Diagrams as Interaction Model

Contrary to StateCharts which focus on internal behaviors of components, UML Sequence Diagrams express in an intuitive and visual way how system components interact and communicate to provide system level functionality.

Recently, many established features of high level MSC [10] have been integrated into the version 2.0 of UML [15], namely interaction operators among fragments and adoption of partial order among interaction events rather that the related messages. Each interaction fragment alone is a partial view of the system behavior but when combined all together by means of the new interaction operators, interactions provide a more complete system description.

However their semantics remain described in natural language with sometimes OCL formulas. Accordingly, to allow using automated tools for their analysis, simulation and verification, sequence diagrams should be given a precise and formal semantics by means of rigorous mathematical formalisms [8,20,22,4,11].

Many approaches are proposed to assign a formal semantics to UML interaction diagrams. Two kinds of mathematical models are used to achieve those definitions: similarly to UML standard specification [15]. Most of models are based on traces of events occurrences related to exchanged messages between different components [8,20]. Whereas models of the second family have a branching time semantics: Automata [22,4], events structures [11] and lattice-like graphs [7] which record faithfully the intended traces of events together with choices possibilities.

In this context, we customize below a formal approach we have proposed in [7] to define branching time semantics for UML Sequence Diagrams in denotational style. It deals with the partial order and yields transition systems that specify faithfully the intended behaviors by recording both traces of all interaction components together with branching. Therefore we provide our mathematical structure with few generalized algebraic operations making it easy to give formal definitions of interaction operators in a compositional manner.

5.1 Syntactical Aspects of Sequences Diagrams

In a sequence diagram (see fig.4), many participants (components, objects...) may be involved in the interaction where dispatched messages are ordered from top to bottom so that the control flow over time is shown in a clear manner. Each message is defined by two events: message emission and message reception and events situated on the same lifeline are ordered from top to down [15].

Accordingly a message defines a particular communication among communicating entities. This communication can be “raising a signal”, “invoking an operation”, “creating” or “destroying an instance”. The message specifies not only the kind of communication but also the sender and the receiver. The various kinds of communication involved in distributed systems are considered in UML sequence diagrams. Hence messages may be either synchronous or asynchronous.

Besides this, UML 2.0 sequence diagram introduces the notion of basic interaction fragment only represents finite behaviors without branching (when executing a sequence diagram, the only branching is due to interleaving of concurrent events), but these can be composed to obtain more complete descriptions. Basic interaction fragments can be composed in a composite interaction fragment called combined interaction or combined fragment using a set of operators called interaction operators. The unary operators are OPT, LOOP, BREAK and NEG. The others have more than one operand, such as
ALT, PAR, SEQ\textsubscript{weak} and SEQ\textsubscript{strict}. Recurrently the combined fragments can be combined themselves together until obtaining a more complete diagram sequence [15].

Finally interactions in UML 2.0 sequence diagrams are considered as collections of events instead of ordered collections of messages as in UML 1.x.

5.2 A Formal Model for Interaction Behavior

The final graph we derive from any sequence diagram is a labeled transition system \( G = \langle Q, \rightarrow, \Sigma, q_0 \rangle \) where:
- \( Q \) is a set of nodes modeling states (configurations) through which the system goes.
- \( q_0 \) is the initial node of the graph.
- \( \rightarrow \subseteq Q \times \Sigma \times Q \) is a set of transitions between nodes.
- \( \Sigma \) is a set of events denoting sending and receiving messages.

Over transition systems we use two associative algebraic operations: choice (\( \oplus \)), synchronization product (\( \otimes \)) and one unary operation \( * \) called star [7].

As we follow a component-based approach, we select only a subset of sequences diagrams (SD\([C_0]\)), namely those containing the main component \( C_0 \).

Each sequence diagram \( sd \in SD[C_0] \) is built by combining its interactions fragments (ItF) by means of a set (ItO) of interactions operators. Thus, \( sd \) can formally be represented by an abstract structure (ItF, ItO) with ItO = \{alt, opt, par, seq\textsubscript{weak}, seq\textsubscript{strict}, loop, break\} where:
- Interactions operators from \{alt, opt, break\} perform respectively choices between behaviors of two interaction fragments, between one fragment and nothing or between a nested fragment with the remainder of the enclosing fragment.
- seq\textsubscript{weak} and seq\textsubscript{strict} operators are respectively weak and strict version of sequencing between behaviors of two interaction fragments.
- Loop makes it possible to repeat the behavior of the operand many times.

The construction of the transition system of \( sd \) is achieved by combining graphs related to the involved fragments (see fig.5). Those graphs themselves are recurrently built by combining models of their interaction fragments and so on [7].

Let \( \langle . \rangle \) be a semantics function that maps interaction fragments into transition systems in denotational style:
- First we assign to each participant lifeline \( P \) its graph \( \langle P \rangle \) by extracting the tree depicting sequences of ordered events that are sent or received on the lifeline of \( P \).
- Then for any basic interaction fragment \( X \), we build its graph \( \langle X \rangle \) by synchronization conjunction of basic graphs related to involved participants \( \{P_i\}_{i \in \{1...n\}} \).

\[
\langle X \rangle = \langle P_1 \rangle \otimes \ldots \otimes \langle P_n \rangle
\]
- Afterwards, we achieve recurrently various combinations among yielded graphs with regards to combinations over fragments until we obtain the final transition system related to sd.

- Let Y be an interaction fragment obtained by combining two fragments $Y_1$ and $Y_2$ by means of an interaction operator $op_{sd} \in \text{ItO}$. Let $op_G$ be a graph operation that is the mapping of $op_{sd}$ by $[\,\,]$.

$$[Y] = [[Y_1 \op_{sd} Y_2]] = [[[Y_1],] \op_G [[Y_2]]]$$

The mapping of any interaction operator from $\{\text{alt, opt, break}\}$ is a choice operation $\oplus$ of the related graphs. However each interaction operator from $\{\text{par, seqs, seqw}\}$ is mapped in some parameterized version of the synchronization product $\otimes$ which should take care of orderings among events in the specific interleaved sequences. The last operator loop is obviously mapped into the star operation on graphs.

**Remark.** The unary operator NEG used on a combined fragment means that the graph related to the operand depicts an invalid behavior the system should discard.

### 6. Simulation Relations

Given some system under design, once behavioral graphs of its sequence and state diagrams are built, we can proceed to consistency checking between them by setting up an adequate preorder (simulation) relation \cite{17} from sequence diagram into state diagram. We do not deal with equivalence relations because mostly only a subset of requirements are depicted in terms of scenarios which elicit only a subset of prefixes of traces we can extract from the related state diagram.

**Notations.** Let $\sum$ be a vocabulary of symbols, $\sum^*$ is the set of all finite words over $\sum$ including the empty word $\epsilon$. Let $w, w^* \in \sum^*, w \leq w^*$ if $\exists w^* \in \sum^*; w^* = w \cdot w^*$ ($w \cdot w^*$ denotes the concatenation of $w$ and $w^*$).

**Definition 2.** Let $G$ be a transition system $G = (Q, \rightarrow, \sum, q_0)$

- For $p \in Q$, $l(p) = \{a \in \sum \mid a \rightarrow p\}$ is the set of initial actions of $p$.
- If $l(p) = \emptyset$, the state $p$ is called a **deadlock state**.
- We extend the transition relation as follows: $p \rightarrow w \rightarrow q$ where $w \in \sum^*$

  If $w = \epsilon$ then $p = q$ else $w = a_1 \ldots a_n$ such that $p \rightarrow w \rightarrow q$ is a chain of labeled transitions of the form: $p \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow q$.

- The set of traces is $\text{Traces}(p) = \{ w \in \sum^* / p \rightarrow w \rightarrow q \mid l(q) = \emptyset \}$.
- The set of traces of the graph $G$ is $\text{Traces}(G) = \text{Traces}(q_0)$.
- The set of prefixes of traces of $G$: $\text{PTr}(G) = \{ w \in \sum^* / \exists w^* \in \text{Traces}(G) \text{ and } w \leq w^* \}$.

**Definition 3.** The set of readies relating to a state $p$ is:

$$\text{Readies}(p) = \{ [w, X] / \exists p \rightarrow w \rightarrow q \text{ and } l(q) = X \}$$
This set contains sequences of actions that can be performed from a node together with the set of available actions.

The set of readies relating to a graph $G$ is $\text{Readies}(G) = \text{Readies}(q_0)$.

Now we present a preorder relation between transition systems: $G_i = (Q_i, \rightarrow, \sum, q_{i0}) | i = 1, 2$.

**Definition 4.** A simulation is a binary relation $R \subseteq Q_1 \times Q_2$, satisfying, for $a \in \sum$

- If $(p, q) \in R$ and $\forall p' : p \xrightarrow{a} p' \Rightarrow \exists q' : q \xrightarrow{a} q'$ and $(p', q') \in R$.

A node $p$ can be simulated by $q$, notation $p \leq q$, if there is a simulation $R$ with $(p, q) \in R$.

Additionally, $G_1 \leq G_2$ if $q_{01} \leq q_{02}$ (root nodes related).

Note that if $G_1 \leq G_2$ then $\forall [w, X] \in \text{Readies}(G_1)$, there exists $[w, Y] \in \text{Readies}(G_2)$ such that $X \subseteq Y$. Hence the capabilities of first graph are preserved by the second graph.

**Property 2.** The simulation among transition systems is reflexive and associative. Hence, it is a preorder relation.

**Lemma 2.** Let $C_i$ be any component of a system $S$. $S[C_i]$ denotes the StateChart of $C_i$. $\forall C_i \in S : S[C_i] \leq S$.

Proof. Let $RG[C_i]$ be the synchronization product of all components but $C_i$. According to property 1, $RG[S] = RG[C_i] \otimes RG[C_i]$. Moreover following lemma 1, when synchronizing these two reachability graphs, each node from $RG[C_i]$ is combined with the initial node of $RG[C_i]$. So each path of $RG[C_i]$ is a prefix of some path in $RG[S]$. Hence $S[C_i] \leq S$.

**Definition 5.** Let $RG[S]$ be the reachability graph of the state diagram $S$ and $(RG[V], RG[I])$ be the pair of valid and invalid graphs of a set SD of sequence diagrams. We say that $S$ satisfies SD ($S \models SD$) if there exists a preorder $\leq$ such that:

- $\forall sd \in SD, RG[V][sd] \leq RG[S]$ and $\text{Traces}(RG[I][sd]) \not\subseteq \text{PTr}(RG[S])$.

The statement "$G_1[sd] \leq G[S]$" implies that any node in the valid graph of $sd$ can be simulated by some node in the graph of $S$ in such a way any possible action at the first node remains also available in the second one and any reached node from the first node by any action "a" is also simulated by a reached node from the second one by the same action "a".

The statement "$\text{Traces}(G_1[sd]) \not\subseteq \text{PTr}(G[S])$" means that any invalid sequence of actions in $sd$ can not be achieved by the state diagram.

**Theorem 1.** Let $SD[C_i]$ be a subset SD consisting in sequence diagrams related to a component $C_i$. Let $S[C_i]$ denote the StateChart depicting this component $C_i$.

- $\forall C_i \in S : S[C_i] \models SD[C_i] \Leftrightarrow S \models SD$

Proof. ($\Rightarrow$) Following lemma 2, we have $\forall C_i \in S : S[C_i] \leq S$.

As the ready simulation is transitive, we obtain: $\forall C_i, \forall sd \in SD[C_i] : sd \leq S[C_i] \leq S$.

Hence $SD[C_i] \leq S$.

For invalid traces, we know from lemma 1 that $RG[S]$ is isomorphic to the synchronization product of $RG[C_i]$. Therefore, the union of execution paths of $RG[C_i]$ gives the same set of paths of $S$. if there is a sequence in $G_1[sd]$ which exists in $\text{Traces}(S)$, it belongs particularly to some component $C_i$. However this is impossible because for each $C_i$ the hypothesis stipulates that $\text{Traces}(G_1[sd]) \not\subseteq \text{PTr}(G[C_i])$.

($\Leftarrow$) We assume there is some component $S[C_i]$ which does not fulfill one sequence diagram $sd$ in which $C_i$ is involved. So there is a path in $sd$ which contains at least one action from this component and which is discarded in $S[C_i]$. Therefore when synthesizing $R[S]$ from its components and following the isomorphism lemma, we should not find the discarded path in the whole graph. However $S$ fulfills all sequence diagrams. Hence our supposition is false.

The same idea is used to prove that any invalid Trace in $SD[C_i] \not\subseteq \text{PTr}(G[C_i])$.

**7. Conclusion**

In this paper, we have presented a modular methodology for consistency checking between UML dynamic diagrams in order to avoid the state explosion problem. We have described an approach of extracting partial reachability graphs from state diagrams with regards to some component. Likewise, we sum-
marized a method of extracting partial behavioral graphs from the set of sequence diagrams where this component is involved as participant. Afterwards, we have given a preorder relation to check whether our reachability graph can simulate the second graphs.

This methodology seems practical to substitutability checking of components and may be also extended to find out whether and how temporal constraints on events paths could be fulfilled with respect to time intervals on arcs of reachability graphs.

References

1. P.C. Attie. Synthesis of large concurrent programs via pairwise composition in Proc. of CONCUR’99: 10th International Conference on Concurrency Theory