JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 15, No. 4, December 2021, pp. 148-159

Structure-Behavior Coalescence Method for Formal Specification of UML 2.0 Sequence Diagrams

Steve Haga and William Shanchung Chao
Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
stevewhaga@cse.nsysu.edu.tw, architectchao@gmail.com

Wei-Ming Ma*
Department of Information Management, Cheng-Shiu University, Kaohsiung, Taiwan
k3666@gcloud.csu.edu.tw

Abstract
This paper presents the structure-behavior coalescence (SBC) method for the formal specification of UML sequence diagrams. The SBC method, also named SBC sequence diagrams (SBC-SqD), includes syntax and semantics. The syntax allows the hierarchy of a sequence diagram to be represented as a parse tree. The semantic meaning of that sequence diagram is then represented by its message sending-and-receiving event transition graph (MSRETG). In order to create this MSRETG, the semantic meaning of sequence diagram (SD) modules is presented, including the meaning of these modules when they are used in a hierarchy that defines another module. Such hierarchical decomposition is formally provided for all sequence diagram modules, including all types of loops. In UML, all SD modules are also called combined fragments (CF). The obtained MSRETG has advantages over previous formal specifications, because it is a fully-complete specification and is compact and readable.

Category: Compilers / Programming Languages

Keywords: Structure-behavior coalescence; Sequence diagram; Message sending-and-receiving event transition graph; Combined fragment

I. INTRODUCTION

Sequence diagrams (SDs) are a major aspect of Unified Modeling Language (UML) specifications [1, 2]. Unlike other UML diagrams, SDs focus on the sequence ordering of events, which are modeled as messages passed between a system's component objects. The sequencing of events may be either strict or weak. A weak sequence only specifies the order in which messages are sent, and not the order in which they are received. SDs represent these two types of sequences as strict or weak sequence module blocks. There are also modules to specify parallelism, alternative options, and loops.

A module in a system's SD description may contain other modules, so complexity is managed in hierarchical layers. To extract the semantic meaning of an SD specification, rules are needed for *Composition* across the layers of this hierarchy into a single unified view of the full-system behavior.

In practical use within industry, the adoption of the

Open Access http://dx.doi.org/10.5626/JCSE.2021.15.4.148 http://jcse.kiise.org

Received 8 October 2021; **Accepted** 13 December 2021
*Corresponding Author

UML model to system design was promoted and guided by the Object Management Group (OMG) [3, 4]. The current guidance does not contain a fully-formal specification. Users have some freedom in how they draw SDs and in what it means to compose them. In other words, human common sense is used to fill in the gaps in the specification. Although this approach works well-enough in practice, it suffers from two problems: (1) it creates the potential for a miscommunication between design teams, who may not all recognize the common-sense assumptions of their system specification, and (2) it hinders the development of executable UML for sequence diagrams.

Various approaches have sought to address the need for formalism with a theoretical foundation [5-12]. In some of these works, the *syntax* of an SD was represented as a sentence in a grammar, and the *semantics* of an SD was represented as the set of all possible execution sequences that may occur. Unfortunately, the underlying grammar was not always formally specified, and the semantics of *Composition* across the hierarchical layers of the SD was not covered, particularly for loop composition.

In this paper, a grammar of the SBC method is proposed, which is fully expressive, while also being more precise than previous grammars. For example, some less-precise grammars do not consider as a syntax error the illogical sequence where an event follows the terminate signal. Any SD can be represented as a sentence in our language, with the syntax tree for its sentence corresponding to the hierarchical construct of that SD.

In Section II, an overview of UML sequence diagrams is provided. In Section III, the syntax and semantics of our structure-behavior coalescence approach to SDs is presented. In Section IV, the benefits of this approach are compared to current approaches. In Section V, conclusions are drawn.

II. UML 2.0 SEQUENCE DIAGRAMS

In UML 2.0, the structural elements of the object are represented by lifelines on the sequence diagram. The sequence diagram describes the interactions between these lifelines as an ordered sequence of occurrence specifications, which describe different types of events, such as the sending and receiving of messages.

In order to model ordering of event occurrences more complex than simple sequences, UML sequence diagrams used special modules called combined fragments (CF). Combined fragments have an operator and a set of operands, which themselves can be combined fragments, thus forming a tree of combined fragments. The ordering operators that describe different ordering semantics are: inactive, weak and strict sequencing, loop, alternative, and parallel.

Inactive combined fragment: The UML 2.0 sequence diagrams include an inactive combined fragment, shown as "◉", in which there are no interaction points.

Weak sequencing of messages: An interaction fragment consists of one or more messages. UML 2.0 sequence diagrams use the weak sequencing operator (**seq**) to implement the weak sequencing of messages. Fig. 1 shows that the interaction fragment "IF_{01}" will execute the messages "m_{01}", "m_{02}", and "m_{03}" sequentially in a weak manner.

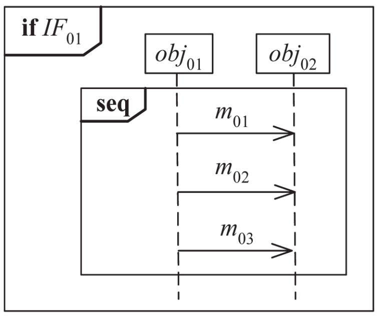

Fig. 1. Weak sequencing of messages.

Strict sequencing of interaction fragments: UML 2.0 sequence diagrams used the strict sequencing operator (**strict**) to implement the strict sequencing of interaction fragments. Fig. 2 shows that the UML 2.0 sequence diagram represented by the combined fragment "CF_{11}" defined as "IF_{11} **strict** IF_{12} **strict** CF_{12}" will first execute the interaction fragments "IF_{11}" and "IF_{12}" sequentially in a strict manner and then continue to execute the combined fragment "CF_{12}".

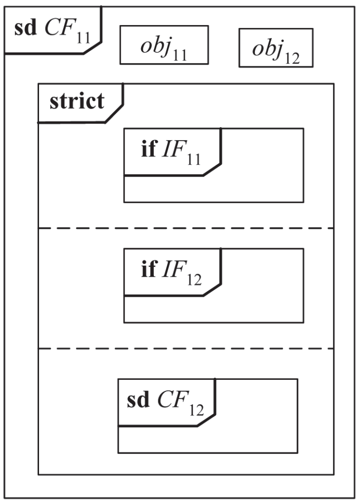

Fig. 2. Strict sequencing of interaction fragments.

Loop definition of combined fragments: UML 2.0 sequence diagrams use the loop operator (**loop**) to implement the loop definition of combined fragments. A loop may define lower and upper bounds on the number of iterations as well as the guard condition. Fig. 3 shows that the UML 2.0 sequence diagram represented by the combined fragment "CF_{21}" defines the messages "m_{21}" and "m_{22}" to be executed repeatedly until its termination constraint was met.

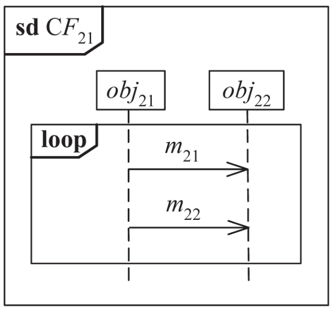

Fig. 3. Loop definition of a combined fragment.

Reference definition of combined fragments: UML 2.0 sequence diagrams used the reference operator ($\overset{\text{ref}}{=}$) to implement the reference definition of combined fragments. Fig. 4 shows that the combined fragment "CF_{21}" was referenced.

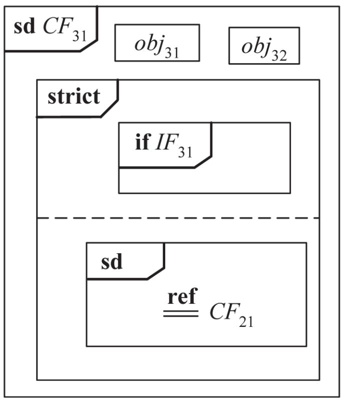

Fig. 4. An example of reference definitions for combined fragments.

Alternative composition of combined fragments: UML 2.0 sequence diagrams used the alternative operator (**alt**) to implement the alternative *Composition* of combined fragments, written as "CF_1 **alt** CF_2", indicating that the alternative operator will execute "CF_1" and "CF_2" alternately. Fig. 5 shows that the UML 2.0 sequence diagram represented by the combined fragment "CF_{41}" was alternately composed of the combined fragments "CF_{42}" and "CF_{44}".

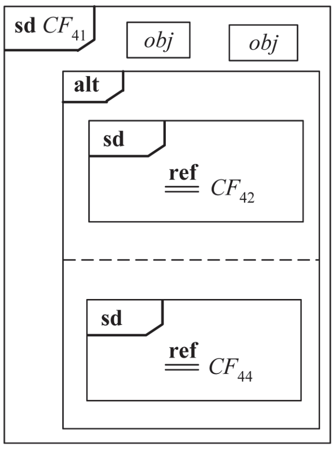

Fig. 5. Alternative definition of combined fragments.

Parallel composition of combined fragments: UML 2.0 sequence diagrams used the parallel operator (**par**) to realize the parallel *Composition* of combined fragments, written as "CF_1 **par** CF_2", indicating that the parallel operator interleaved its operands independently and concurrently. Fig. 6 shows that the UML 2.0 sequence diagram represented by the combined fragment "CF_{51}" was composed of the combined fragments "CF_{42}" and "CF_{44}" in parallel.

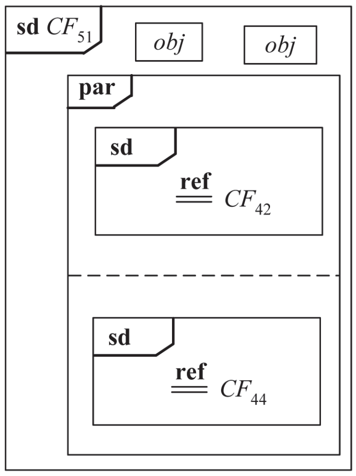

Fig. 6. Parallel definition of combined fragments.

III. STRUCTURE-BEHAVIOR COALESCENCE METHOD

The structure-behavior coalescence method for UML 2.0 sequence diagrams was named SBC sequence diagrams (SBC-SqD). In SBC-SqD, each combined fragment was regarded as a process.

A. Entities of SBC-SqD

Table 1 presents the entities of SBC-sequence diagram descriptions. These entities have some similarity to the process algebra entities of SBC system modeling [13, 14], but these new entities define weak sequence interactions and also used guard conditions to control loops, rather than provided a specific loop entity (i.e., there is no χ entity). As shown in Table 1, we used C to represent a set of guard conditions, and c_1, c_2... to represent the elements of C. Further, we let Δ be a set of interactions, and employed a_1, a_2... to range over Δ. We assumed R be the set of prefixes, and employed r_1, r_2... to range over R. We let Ψ be the set of combined fragments, and employed CF_1, CF_2... to range over Ψ. Further, we assumed Φ be the set of combined fragment constants, and employed A_1, A_2... to range over Φ.

Table 1. Entities of SBC-SqD

Entity set	Entity name	Entity type
C	c_1, c_2...	guard conditions
SE	snd_m_1, snd_m_2...	message sending events
RE	rcv_m_1, rcv_m_2...	message receiving events
M	m_1, m_2...	messages
E	e_1, e_2...	events
Π	π_1, π_2...	code snippets
R	r_1, r_2...	prefixes
Q	q_1, q_2...	message prefixes
Ψ	CF_1, CF_2...	combined fragments
Φ	A_1, A_2...	combined fragment constants

B. Prefix in SBC-SqD

SBC-SqD is a labelled transition system (LTS) [15-18] which provides a single diagram for UML 2.0 to unify structural and behavioral constructs. In the SBC-SqD transition system, each transition was labelled with a prefix defined as follows.

Definition (message sending-and-receiving event). A message sending-and-receiving event $MSRE = (SE, RE, M, E)$ consists of

- a finite set SE of sending events, and $snd\_m \in SE$,
- a finite set RE of receiving events, and $rcv\_m \in RE$,
- a relation M of messages where $M \subseteq SE \times RE$, and $(snd\_m, rcv\_m) \in M$,
- a union E of events where $E \subseteq SE \cup RE$, and $e \in E$.

Definition (message prefix). A message prefix $MPX = (C, M, \Pi, Q)$ consists of

- a finite set C of optional guard conditions,
- a relation M of messages,
- a finite set Π of optional code snippets,
- a relation $Q \subseteq C \times M \times \Pi \times C \times M \times \Pi$, and $(snd\_c, snd\_m, snd\_\pi, rcv\_c, rcv\_m, rcv\_\pi) \in Q$.

Definition (prefix). A prefix $PX = (C, E, \Pi, R)$ consists of

- a finite set C of optional guard conditions,
- a finite set E of events,
- a finite set Π of optional code snippets,
- a relation $R \subseteq C \times E \times \Pi$, and $(c, e, \pi) \in R$.

In SBC-SqD, all prefixes were guarded by either an explicitly given condition, or the implicit condition [TRUE]. If the value of the condition was TRUE and the message sending and receiving related event was ready, the transition was triggered. Once the transition was triggered, the appended code snippet was executed.

C. Syntax of SBC-SqD

As a formal language, SBC-SqD was syntactically specified by the Backus-Naur form (BNF) grammar, as shown in Fig. 7.

(1) CF ::= ◉

(2) $CF ::= IF$ **strict** CF_1

(3) $IF ::= q$ **seq** IF_1

(4) $IF ::= q$

(5) $CF ::= (CF_1$ **alt** $CF_2)$

(6) $CF ::= (CF_1$ **par** $CF_2)$

(7) CF ::= **loop** $MSRETG$

(8) $CF ::= \overset{\textbf{ref}}{=\!=} A$

Fig. 7. The BNF grammar of SBC-SqD.

Rule 1 allowed the inactive combined fragment expression "◉" to be a valid SBC-SqD combined fragment.

Rule 2 described that the combined fragment "IF **strict** CF_1" will execute the interaction fragment "IF" first and then continue to execute the combined fragment "CF_1". This rule has the desirable property of forcing weak sequencing to be self-contained within its parent strict-sequence block, thus clearing up confusion about whether a weak sequence must finish before entering a new CF. Rule 2 has the added benefit of making the grammar produce an unambiguous parse tree.

Rules 3 and 4 described that the weak sequence composition was sequentially composed of one or more message prefixes "q".

Rule 5 described that the combined fragment expression "CF_1 **alt** CF_2" represents an alternative of whether to execute the combined fragment "CF_1" or "CF_2".

Rule 6 described that the combined fragment expression "CF_1 **par** CF_2" will interleave the combined fragments "CF_1" and "CF_2" independently and concurrently.

Rule 7 described the combined fragment, denoted by the "**loop** *MSRETG*" combined fragment expression, a loop definition of infinite behavior represented by the message sending-and-receiving event transition graph "*MSRETG*," was a valid SBC-SqD combined fragment.

Rule 8 described the combined fragment constant "A" which will be referenced and written as "$\overset{\textbf{ref}}{=\!=} A$".

Additionally, Rule 8 allowed the grammar to be Turing-complete because the **loop**, **par**, and **alt** operators all allowed nesting of further operators within their bodies.

D. MSRETG Overview Diagram

To be grammatically accurate in SBC-SqD, each MSRETG must adhere to the BNF grammar. We used the MSRETG overview diagram (MOD) to represent a grammatically correct MSRETG.

For example, Fig. 8 shows a MSRETG overview diagram with the definition:

$CF_{34} \overset{\textbf{def}}{=\!=} ((\overset{\textbf{ref}}{=\!=} CF_{11})$ **par** $(\overset{\textbf{ref}}{=\!=} CF_{21}))$ **par** $((\overset{\textbf{ref}}{=\!=} CF_{31})$ **alt** $(\overset{\textbf{ref}}{=\!=} CF_{41}))$

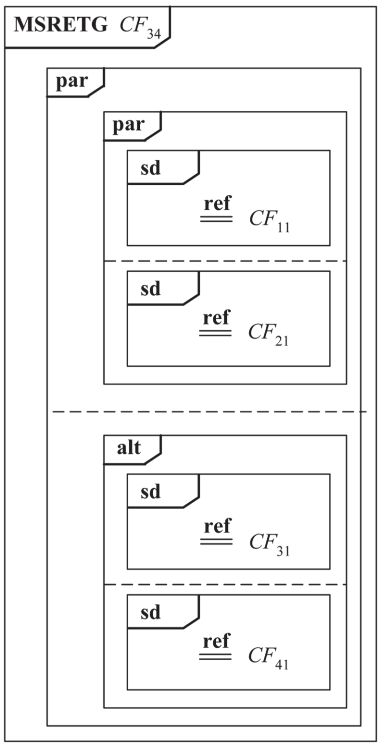

Fig. 8. An example of MSRETG overview diagram.

E. Transitional Semantics of SBC-SqD

In order to give SBC-SqD meaning, we used the MSRETG as a single diagram to specify the semantics of the system.

Definition (message sending-and-receiving event transition graph). A message sending-and-receiving event transition graph $MSRETG = (\Psi, (\pi_0, CF_0), R, MSRETGR)$ consists of

- a finite set Ψ of combined fragments,
- an optional code snippet π_0 in the initial transition, and $\pi_0 \in \Pi$,
- an initial combined fragment $CF_0 \in \Psi$,
- a finite set R of prefixes,
- a transition relation $MSRETGR \subseteq \Psi_1 \times R \times \Psi_2$, where $(CF_j, r, CF_k) \in MSRETGR$ is written as $CF_j \xrightarrow{r} CF_k$.

For example, the message sending-and-receiving event transition relation $MSRETGR_{81} = \{(CF_{81}, (nil, snd\_m_{81}, nil), CF_{82}), (CF_{81}, (nil, snd\_m_{82}, \text{A} = \text{A} + 10;), CF_{83})\}, (CF_{82}, (\text{A} > 0, rcv\_m_{81}, nil), ◉), (CF_{83}, (nil, rcv\_m_{82}, nil), ◉)\}$ that constituted the message sending-and-receiving event transition graph $MSRETG_{81} = (\Psi, ((\text{A} = 20;), CF_{81}), R, MSRETGR_{81})$ is shown in Fig. 9.

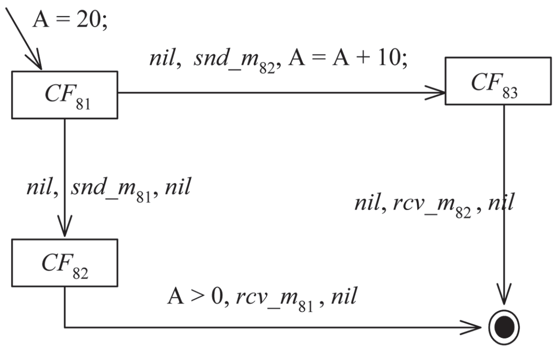

Fig. 9. Message sending-and-receiving event transition graph $MSRETG_{81}$.

In the message sending-and-receiving event transition graph $MSRETG_{81}$, the combined fragment expression is denoted by a rectangle containing its name; the transition from the source combined fragment to the target combined fragment is denoted by an arrow and labelled with a guard, an event and a code snippet; the initial combined fragment (for example, "CF_{81}") is the target combined fragment of the transition that has no source combined fragment; the transition without a source combined fragment is called the initial transition. In $MSRETG_{81}$, the code snippet "A = 20;" was attached to the initial transition. In a combined fragment, if multiple transitions were to be triggered, the choice of trigger will be arbitrary and fair.

In the message sending-and-receiving event transition graph, whenever $CF \xrightarrow{r_1} \cdots \xrightarrow{r_n} CF'$, we called $(r_1 \ldots r_n,$

CF') a derivative of *CF*. For the initial combined fragment CF_0, if $CF_0 \xrightarrow{r_1} \cdots \xrightarrow{r_n} CF_0$ occurred, then the message sending-and-receiving event transition graph is a **loop** definition of the initial combined fragment CF_0. For example, the message sending-and-receiving event transition relation $MSRETGR_{42}$ = {(CF_{42}, (f_count > 0, snd_m_{42}, f_count = f_count - 1;), CF_{43}), (CF_{43}, (*nil*, rcv_m_{42}, *NIL*), CF_{42}), (CF_{42}, (f_count <= 0, snd_m_{43}, *nil*), ◉)} constituted the message sending-and-receiving event transition graph $MSRETG_{42}$ = (Ψ, ((f_count = 100;), CF_{42}), *R*, $MSRETGR_{42}$), as shown in Fig. 10. In $MSRETG_{42}$, $CF_{42} \xrightarrow{r_{42}} \cdots \xrightarrow{r_{42}} CF_{42}$ occurred for the "CF_{42}" initial combined fragment, therefore $MSRETG_{42}$ represented the **loop** definition of the "CF_{42}" combined fragment, that is, CF_{42} = **loop** $MSRETG_{42}$.

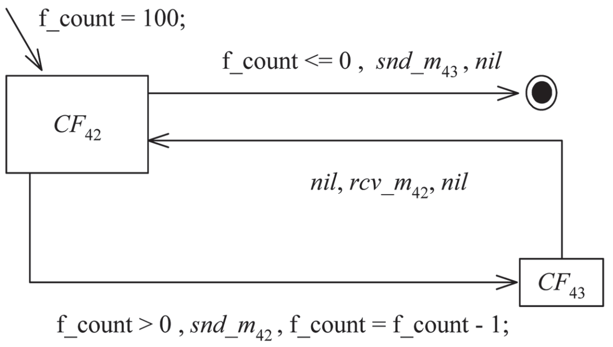
f_count = 100;
f_count <= 0 , snd_m43 , nil
CF42
nil, rcv_m42, nil
CF43
f_count > 0 , snd_m42 , f_count = f_count - 1;

Fig. 10. Loop definition of $MSRETG_{42}$.

For another example, the message sending-and-receiving event transition relation $MSRETGR_{44}$ = {(CF_{44}, (g_count > 0, snd_m_{44}, g_count = g_count - 1;), CF_{45}), (CF_{45}, (*nil*, rcv_m_{44}, *NIL*), CF_{44}), (CF_{44}, (g_count <= 0, snd_m_{45}, *nil*), ◉)} constitutes the message sending-and-receiving event transition graph $MSRETG_{44}$ = (Ψ, ((g_count = 200;), CF_{44}), *R*, $MSRETGR_{44}$), as shown in Fig. 11. In $MSRETG_{44}$, $CF_{44} \xrightarrow{r_{44}} \cdots \xrightarrow{r_{44}} CF_{44}$ occurred for the "CF_{44}" initial combined fragment, therefore $MSRETG_{44}$ represented the **loop** definition of the "CF_{44}" combined fragment, that is, CF_{44} = **loop** $MSRETG_{44}$.

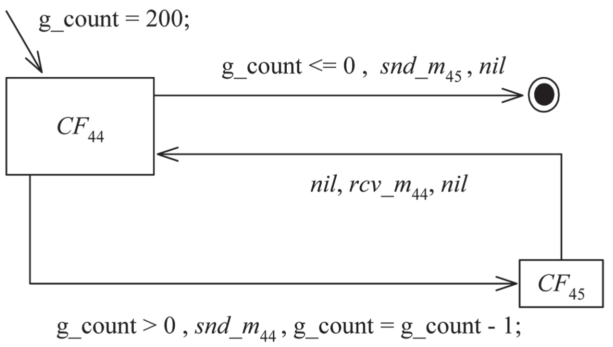
g_count = 200;
g_count <= 0 , snd_m45 , nil
CF44
nil, rcv_m44, nil
CF45
g_count > 0 , snd_m44 , g_count = g_count - 1;

Fig. 11. Loop definition of $MSRETG_{44}$.

F. Transitional Semantics of Composition

SBC-SqD semantics consist of defining each MSRETGR, which is associated with the transition rules of combined fragment operators. These transition rules followed the construct of combined fragment expressions. Fig. 12 gives the complete set of transition rules: Strict Sequence, Alternative, and Parallel Composition.

Strict
$$\frac{}{(IF\ \mathbf{strict}\ CF_1) \overset{IF}{\dashrightarrow} CF_1}$$

Alternative$_1$
$$\frac{CF_1 \xrightarrow{r} CF_1'}{(CF_1\ \mathbf{alt}\ CF_2) \xrightarrow{r} CF_1'}$$

Alternative$_2$
$$\frac{CF_2 \xrightarrow{r} CF_2'}{(CF_1\ \mathbf{alt}\ CF_2) \xrightarrow{r} CF_2'}$$

Parallel$_1$
$$\frac{CF_1 \xrightarrow{r} CF_1'}{(CF_1\ \mathbf{par}\ CF_2) \xrightarrow{r} (CF_1'\ \mathbf{par}\ CF_2)}$$

Parallel$_2$
$$\frac{CF_2 \xrightarrow{r} CF_2'}{(CF_1\ \mathbf{par}\ CF_2) \xrightarrow{r} (CF_1\ \mathbf{par}\ CF_2')}$$

Fig. 12. Transition Rules for SBC-SqD.

Transition rule of Strict Sequencing: We interpreted the strict sequencing rule as: Under any situation, we generally inferred (*IF* **strict** CF_1) $\overset{IF}{\dashrightarrow} CF_1$. In other words, the combined fragment expression, with an interaction fragment prefixed to it, will use this interaction fragment "*IF*" to accomplish this task of strict sequencing.

"CF_2" is used to define the combined fragment expression "*IF* **strict** CF_1", written as "$CF_2 \overset{\mathbf{def}}{=}$ *IF* **strict** CF_1". Strict sequencing rule updating the initial transition and code snippets on the combined fragments "CF_2" and "CF_1" is described as follows: (1) the (resulting) code snippet in the last transition of "*IF*" will be the concatenation of the (original) code snippet in the last transition of "*IF*" and the code snippet in the initial transition of the combined fragment "CF_1"; (2) the initial transition of the combined fragment "CF_1" will be deleted; (3) the code snippet in initial transition of the combined fragment "CF_2" will be rewritten.

As an example of Strict Sequencing transition rule, if we used "IF_{71}" to define the interaction fragment expression "q_{71} **seq** q_{72}" and "CF_{71}" to define the combined fragment expression "IF_{71} **strict** ($\overset{\mathbf{ref}}{=}$ CF_{42})", written as "$CF_{71} \overset{\mathbf{def}}{=}$ IF_{71} **strict** ($\overset{\mathbf{ref}}{=}$ CF_{42})", then the message sending-and-receiving event transition relation $MSRETGR_{71}$ = {(CF_{71}, (*nil*, *snd_m*$_{71}$, *nil*), *CF*$_{72}$), (*CF*$_{72}$, (*nil*, *rcv_m*$_{71}$, *nil*), *CF*$_{73}$)), (*CF*$_{72}$, (*nil*, *snd_m*$_{72}$, *nil*), *CF*$_{74}$)), (*CF*$_{73}$, (*nil*, *snd_m*$_{72}$, *nil*), *CF*$_{75}$)), (*CF*$_{74}$, (*nil*, *rcv_m*$_{71}$, *nil*), *CF*$_{75}$)), (*CF*$_{75}$, (*nil*, *rcv_m*$_{72}$, f_count = 100;)), *CF*$_{42}$), (*CF*$_{42}$, (f_count > 0, *snd_m*$_{42}$, f_count = f_count - 1;), *CF*$_{43}$), (*CF*$_{43}$, (*nil*, *rcv_m*$_{42}$, *nil*), *CF*$_{42}$), (*CF*$_{42}$, (f_count <= 0, *snd_m43*, *nil*), ◉)} constituted the message sending-and-receiving event transition graph *MSRETG*$_{71}$ = (*Ψ*, (*nil*, *CF*$_{71}$), *R*, *MSRETGR*$_{71}$), as shown in Fig. 13.

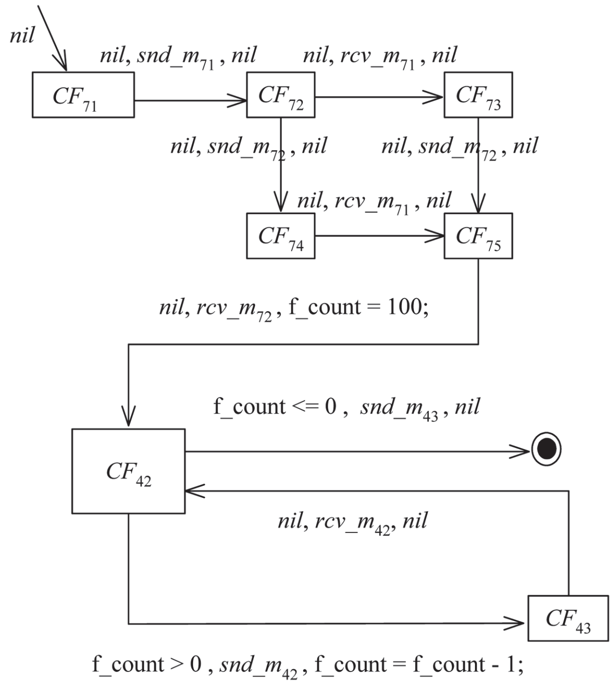

Fig. 13. Message sending-and-receiving event transition graph *MSRETG*$_{71}$.

Transition rule of Alternative Composition: Alternative Composition has two transition rules. Rule Alternative$_1$ showed that from $CF_1 \xrightarrow{r} CF_1'$ we inferred $(CF_1 \textbf{ alt } CF_2) \xrightarrow{r} CF_1'$. Rule Alternative$_2$ showed that from $CF_2 \xrightarrow{r} CF_2'$ inferred $(CF_1 \textbf{ alt } CF_2) \xrightarrow{r} CF_2'$.

"*CF*$_3$" is used to define the combined fragment expression "*CF*$_1$ **alt** *CF*$_2$", written as "$CF_3 \overset{\text{def}}{=} CF_1$ **alt** *CF*$_2$". Alternative Composition updating the initial transition and code snippets on the combined fragments "*CF*$_1$" and "*CF*$_2$" and "*CF*$_3$" is described as follows: (1) the code snippet in the combined fragment "*CF*$_3$" will be the concatenation of the code snippet in the initial transition of the combined fragment "*CF*$_1$" and the code snippet in the initial transition of the combined fragment "*CF*$_2$"; (2) the initial transition of the combined fragment "*CF*$_1$" will be deleted; (3) the initial transition of the combined fragment "*CF*$_2$" will be deleted.

As an example of Alternative Composition transition rule, if we use "*CF*$_{41}$" to define the combined fragment expression "($\overset{\text{ref}}{=}$ *CF*$_{42}$) **alt** ($\overset{\text{ref}}{=}$ *CF*$_{44}$)", written as "*CF*$_{41}$ $\overset{\text{def}}{=}$ ($\overset{\text{ref}}{=}$ *CF*$_{42}$) **alt** ($\overset{\text{ref}}{=}$ *CF*$_{44}$)", then we will obtain the message sending and receiving event transition relation *MSRETGR*$_{41}$ = {(*CF*$_{41}$, (f_count > 0, *snd_m*$_{42}$, f_count = f_count - 1;), *CF*$_{43}$), (*CF*$_{41}$, (f_count <= 0, *snd_m43*, *nil*), ◉), (*CF*$_{41}$, (g_count > 0, *snd_m*$_{44}$, g_count = g_count - 1;), *CF*$_{45}$), (*CF*$_{45}$, (*nil*, *rcv_m*$_{44}$, *nil*), *CF*$_{44}$), (*CF*$_{41}$, (g_count <= 0, *snd_m45*, *nil*), ◉), (*CF*$_{43}$, (*nil*, *rcv_m*$_{42}$, *nil*), *CF*$_{42}$) , (*CF*$_{45}$, (*nil*, *rcv_m*$_{44}$, *nil*), *CF*$_{44}$), (*CF*$_{42}$, (f_count > 0, *snd_m*$_{42}$, f_count = f_count - 1;), *CF*$_{43}$), (*CF*$_{42}$, (f_count <= 0, *snd_m43*, *nil*), ◉), (*CF*$_{44}$, (g_count > 0, *snd_m*$_{44}$, g_count = g_count - 1;), *CF*$_{45}$), (*CF*$_{44}$, (g_count <= 0, *snd_m45*, *nil*), ◉)} constitutes the message sending-and-receiving event transition graph *MSRETG*$_{41}$ = (*Ψ*, ((f_count = 100; g_count = 200;), *CF*$_{41}$), *R*, *MSRETGR*$_{41}$), as shown in Fig. 14.

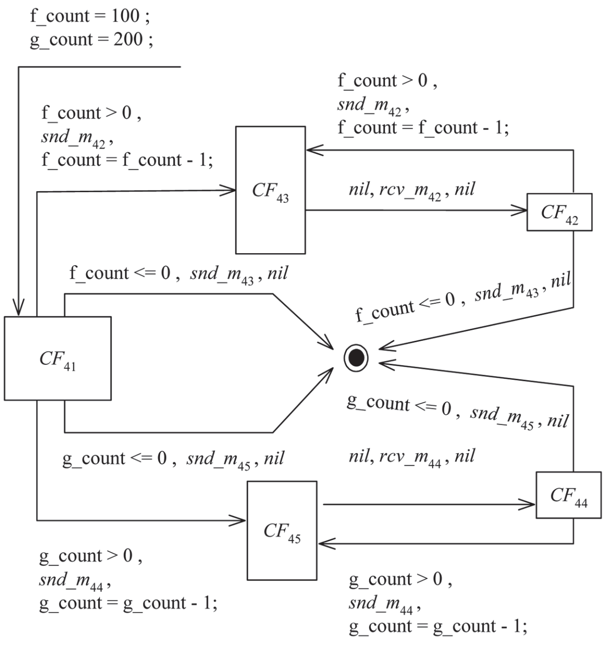

Fig. 14. Message sending-and-receiving event transition graph *MSRETG*$_{41}$.

Transition rule of Parallel Composition: Parallel Composition has two transition rules. Rule Parallel$_1$ showed that from $CF_1 \xrightarrow{r} CF_1'$ we inferred $(CF_1 \textbf{ par } CF_2) \xrightarrow{r} (CF_1' \textbf{ par } CF_2)$. Rule Parallel$_2$ showed that from $CF_2 \xrightarrow{r} CF_2'$ we will infer $(CF_1 \textbf{ par } CF_2) \xrightarrow{r} (CF_1 \textbf{ par } CF_2')$.

"*CF*$_4$" is used to represent the combined fragment expression "*CF*$_1$ **par** *CF*$_2$", written as "$CF_4 \overset{\text{def}}{=} CF_1$ **par** *CF*$_2$". Parallel Composition updating the initial transition and code snippets on the combined fragment "*CF*$_4$" is described as follows: The code snippet in the combined fragment "*CF*$_4$" will be the concatenation of the code snippet in the initial transition of the combined fragment "*CF*$_1$" and the code snippet in the initial transition of the combined fragment "*CF*$_2$".

As an example of Parallel Composition transition rule, if "CF_{51}" is used to represent the combined fragment expression "($\overset{\text{ref}}{=}$ CF_{42}) **par** ($\overset{\text{ref}}{=}$ CF_{44})", written as "CF_{51} $\overset{\text{def}}{=}$ ($\overset{\text{ref}}{=}$ CF_{42}) **par** ($\overset{\text{ref}}{=}$ CF_{44})", then we will obtain the message sending-and-receiving event transition relation $MSRETGR_{51}$ = {(CF_{51}, (f_count > 0, *snd_m*$_{42}$, f_count = f_count - 1;), CF_{43} **par** CF_{44}), (CF_{51}, (f_count <= 0, *snd_m43*, *nil*), ◉ **par** CF_{44}), (CF_{51}, (g_count > 0, *snd_m*$_{44}$, g_count = g_count - 1;), CF_{42} **par** CF_{45}), (CF_{51}, (g_count <= 0, *snd_m45*, *nil*), CF_{42} **par** ◉), (CF_{42} **par** CF_{45}, (f_count > 0, *snd_m*$_{42}$, f_count = f_count - 1;), CF_{43} **par** CF_{45}), (CF_{42} **par** CF_{45}, (f_count <= 0, *snd_m43*, *nil*), ◉ **par** CF_{45}), (CF_{42} **par** CF_{45}, (*nil*, *rcv_m*$_{44}$, *nil*), *nil*), CF_{51}), (CF_{43} **par** CF_{44}, (*nil*, *rcv_m*$_{42}$, *nil*), CF_{51}), (CF_{43} **par** CF_{44}, (g_count > 0, *snd_m*$_{44}$, g_count = g_count - 1;), CF_{43} **par** CF_{45}), (CF_{43} **par** CF_{44}, (g_count <= 0, *snd_m45*, *nil*), CF_{43} **par** ◉), (CF_{43} **par** CF_{45}, , (*nil*, *rcv_m*$_{42}$, *nil*), CF_{42} **par** CF_{45}), (CF_{43} **par** CF_{45}, (*nil*, *rcv_m*$_{44}$, *nil*), CF_{43} **par** CF_{44}), (◉ **par** CF_{44}, (g_count > 0, *snd_m*$_{44}$, g_count = g_count - 1;), ◉ **par** CF_{45}), (◉ **par** CF_{44}, (g_count <= 0, *snd_m45*, *nil*), ◉ **par** ◉), (◉ **par** CF_{45}, (*nil*, *rcv_m*$_{44}$, *nil*), ◉ **par** CF_{44}), (CF_{43} **par** ◉, (*nil*, *rcv_m*$_{42}$, *nil*), CF_{42} **par** ◉), (CF_{42} **par** ◉, (f_count <= 0, *snd_m43*, *nil*), ◉ **par** ◉)} constituted the message sending-and-receiving event transition graph $MSRETG_{51}$ = (Ψ, ((f_count = 100; g_count = 200;), CF_{51}), *R*, $MSRETGR_{51}$), as shown in Fig. 15.

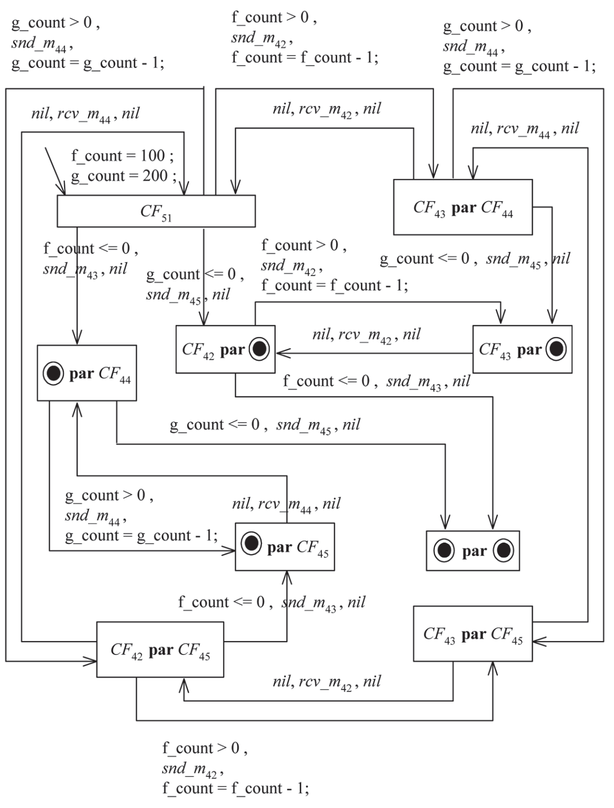
g_count > 0 , snd_m44 , g_count = g_count - 1;
f_count > 0 , snd_m42 , f_count = f_count - 1;
g_count > 0 , snd_m44 , g_count = g_count - 1;
nil, rcv_m44 , nil
nil, rcv_m42 , nil
nil, rcv_m44 , nil
f_count = 100 ; g_count = 200 ;
CF51
CF43 par CF44
f_count <= 0 , snd_m43 , nil
g_count <= 0 , snd_m45 , nil
f_count > 0 , snd_m42 , f_count = f_count - 1;
g_count <= 0 , snd_m45 , nil
CF42 par ◉
nil, rcv_m42 , nil
CF43 par ◉
◉ par CF44
f_count <= 0 , snd_m43 , nil
g_count <= 0 , snd_m45 , nil
g_count > 0 , snd_m44 , g_count = g_count - 1;
nil, rcv_m44 , nil
◉ par CF45
◉ par ◉
f_count <= 0 , snd_m43 , nil
CF42 par CF45
CF43 par CF45
nil, rcv_m42 , nil
f_count > 0 , snd_m42 , f_count = f_count - 1;

Fig. 15. Message sending-and-receiving event transition graph $MSRETG_{51}$.

The complete combined fragment space in the Parallel Composition was the Cartesian product of the two combined fragments. It is correct, however it is not readable. Therefore, sometimes we only displayed the action transition graph without executing the parallel composition transition rule.

For example, the message sending-and-receiving event transition graph "$MSRETGR_{51}$" shown in Fig. 15 executed the parallel composition transition rule on the combined fragment expression "($\overset{\text{ref}}{=}$ CF_{42}) **par** ($\overset{\text{ref}}{=}$ CF_{44})". Contrary to "$MSRETGR_{51}$", the message sending-and-receiving event transition graph "$MSRETGR_{61}$" representing the combined fragment expression "($\overset{\text{ref}}{=}$ CF_{42}) **par** ($\overset{\text{ref}}{=}$ CF_{44})" did not execute the parallel composition transition rule, as shown in Fig. 16.

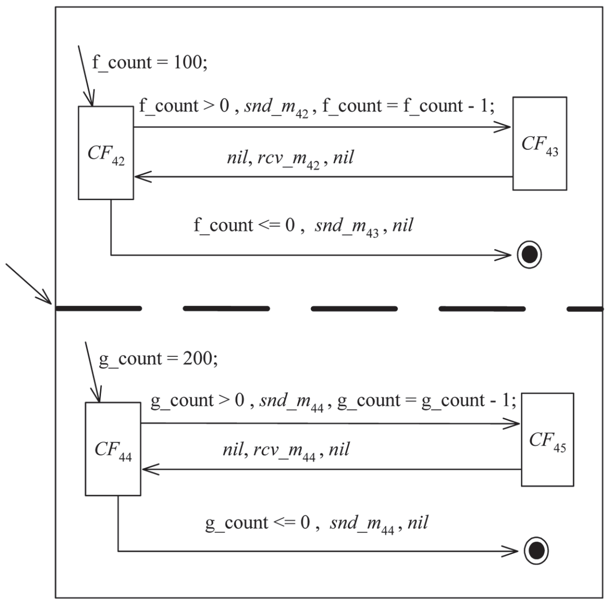
f_count = 100;
f_count > 0 , snd_m42 , f_count = f_count - 1;
CF42
CF43
nil, rcv_m42 , nil
f_count <= 0 , snd_m43 , nil
g_count = 200;
g_count > 0 , snd_m44 , g_count = g_count - 1;
CF44
CF45
nil, rcv_m44 , nil
g_count <= 0 , snd_m44 , nil

Fig. 16. Message sending-and-receiving event transition graph $MSRETG_{61}$.

Although the appearances of the message sending-and-receiving event transition diagrams "$MSRETG_{51}$" and "$MSRETG_{61}$" looked different, they were equivalent in behavior.

IV. RESULTS AND COMPARISON

After composing any specific system's sequence diagrams, a single, unified MSRETG was obtained. This unified MSRETG served as the semantic representation of the top sequence diagram. There were, however, a number of previously proposed semantic representations. All such representations must be semantically equivalent (for valid sequence diagrams). Nonetheless, the differences between representations affected usability. For example, our representation of loops required the end user to describe them as MSRETGs; and this was only possible because the MSRETG description of loops was intuitive. Another advantage of an intuitive representation was that it can help users to understand and to debug their system interactions.

Four alternative semantic representations were typical of the related work. In this section, a representative paper of each of these four representations is compared against our proposed approach. The semantic representations of these four related works are described as follows.

First, Haugen et al. [5] used *traces* to specify semantics. In this approach, the event behavior of a system was described as the set of all possible event orderings that could occur within that system. Each unique event ordering was a single trace. For example, the sentence "q_1 **seq** q_2 **strict** ◉" has two possible event orderings, depending on whether q_2 was sent before q_1 was received.

Second, Hammal [6] used a *lattice-like* graph to allow the merging after two kinds of branching possibilities: (1) after a weak sequence and (2) after a loop with a fixed number of iterations, *n* (such a loop can be unrolled into a sequence of n combined fragments). The combined fragment CF_1 for such a weak sequence or for such a fixed-iteration loop can be followed, in strict sequence, by any other CF_2. In such cases, the branching possible execution paths within the CF_1 all eventually converge on entering CF_2. Merging the branching trees therefore reduced the storage space compared to the trace representation.

Third, Cengarle and Knapp [7] represented sequence diagram semantics as an *institution* [19]. An institution is an annotated first-order logic description of the system, which allowed for analysis using mathematical tools derived for such institutions. However, the user does not find the depiction comprehensible.

Fourth, Knapp and Wuttke [8] represented sequence diagram semantics with *interaction automata*. Our MSRETG were also *interaction automata*. The format of Knapp's automata was more complex than our MSRETG format, but this was due to the fact that Knapp's automata are only intended to be used for model testing. Our MSRETG format can be directly converted to the format of Knapp's automata, thereby allowing all of the same model-checking techniques to be used.

A. Formal Syntax and Semantics for Sequential Behavior

This paper proposed a syntax and semantics for sequences. Notably, rules 2–4 of our grammar (see Fig. 7) restricted weak interactions to clearly demarcate regions. This is because any chain of weak interactions *IF* must be followed by a strict sequence. As a consequence of this, the grammar of Fig. 7 was right recursive, and therefore unambiguous. As an unambiguous grammar, the parse tree of any sentence can be unique. The sequence diagram was derived from this parse tree, with the nested levels of contained *CF*s corresponding to the levels of the parse tree; the sequence diagram that derived a sentence in our grammar was therefore also unique.

In [5], a grammar was used but not formally specified. The implied grammar suggested that the CF_1 **seq** CF_2 operation was legal. The implications of weak sequencing before an arbitrary combined fragment were not discussed. The semantics of a weak sequence was the set of all possible traces through the weakly-sequenced region.

Similarly in [6], a grammar was used but was not formally specified. The weak **seq** operator was stated to precede an arbitrary combined fragment. The algorithm tried to isolate weak regions into a lattice structure similar to the diamond structure in the upper-right portion of our Fig. 12.

Cengarle and Knapp [7] provided a formal grammar that explicitly allowed **seq** to precede an arbitrary combined fragment. As the goal of this work was only to create an institutional description, the issue of the corresponding semantic meaning was mathematically modelled but not discussed.

Knapp and Wuttke [8] provided a grammar concerned with only creating (perhaps by hand) an interaction automata semantic representation. Since the automata were created by hand, it was not restricted. However, the issue of weak sequence interactions was discussed. It is stated that strict loops (loops which were preceded by a **strict** interaction) might be preferred, and that other loops would need to be restricted in certain ways (by the designer of the interaction automaton for that loop).

Compared to the previous works, few have given an explicit grammar. None of those who have provided a grammar have specifically specified that the parse tree for a phrase should match to its sequence diagram. In fact, there was some debate among the UML 2.0 community over how to properly express a sequence diagram. By introducing for the first time an unambiguous grammar, the proper design of the sequence diagram became clear, which can avoid confusion between users. Moreover, the unambiguous nature of the grammar disallowed nonsensical sentences like "q_1 **strict** ◉ **strict** q_2 **strict** ◉", which other grammars would allow.

B. Formal Syntax and Semantics for Loops

In the literature, there is some confusion about the definition of a loop. As Hammal [6] described, there are two types of the loop: a simple loop for a known finite number of iterations and a potentially unbounded loop. The bounded loop was describable by unrolling it into a plain sequence. Therefore, such a loop did not require our **loop** operator (although it is allowed to use it). Consequently, we will not consider that case further. As for unbounded loops, our grammar treated them as leaf nodes in the sentence. The designer provided an MSRETG for the loop behavior. The loop's MSRETG has combined fragments for its nodes; and combined fragments can be expressed as sentences in our grammar. Therefore, the MSRETG provided for the loop was a glue for connecting the various sentences that defined the system. Although the user can, grammatically, provide any MSRETG, the semantically correct one was carefully designed using guard conditions.

In [5], loops were not discussed. However, despite, they were discussed in [7], it was only for the trivial, bounded case. In [6], unbounded loops were discussed and properly identified as being leaf nodes in the parse tree. No solution was provided for composing these loops. In other words, the individual sentences that described the system were defined, but these cannot be combined to create a single, unified semantic description of the sentence. In [8], unbounded loops were studied. Because this work directly created interaction automata for all components, it was able to create a unified semantic description. The limitation was that the automata were complex and were not derived from a grammar.

Compared to the related work, the introduction of the MSRETG specification was a key advantage. Section III-D described how the grammar converted a state diagram into an MSRETG. Since the loops were also defined in the same MSRETG format, the composition rules of Section III-E worked equally well for them. Figs. 9 and 10 showed how a sequence of events that occurred within a loop can be defined by an MSRETG, which was a formally defined labeled transition system (LTS). Fig. 12 demonstrates the *Composition* of the loop after a sequence, further indicating the ability to fully define the semantic meaning of loops within a larger description.

Other works, which did not use such a simple semantic representation, could not employ this solution. The full semantic meaning was derived in [8] by hand-describing the system. In short, no other solution has been able to seamlessly compose across loops.

C. Formal Syntax and Semantics for Alternative Behavior

The **alt** operator was applied to two combined fragments, CF_1 and CF_2. Notable in our grammar, every combined fragment ended with a "◉". Therefore the grammar was clear that no sequence CF_3 came after that **alt** operator. This restriction did not limit expressivity, however, because the CF_1 and CF_2 fragments may both contain CF_3 within them. Another feature of the **alt** operator is the ability to use guard conditions to determine the choice taken. When a condition is not used, the alternatives are both considered possible, as is standard practice.

All of the related work papers contained an **alt** operator. Some, such as [8] also provided guarded execution. However, not every study made it clear that no combined fragments might appear once the alt operation was completed.

The key limitation of other works was in *Composition*, particularly across loops. If the alt was inside of a loop, or if a loop was inside of an alt, only our approach can collect the semantic information into one graph. Section III-E described the approach to accomplish this.

D. Formal Semantics for Parallel Behavior

The **par** operator was applied to two combined fragments, CF_1 and CF_2. Similar to the **alt** operator, no sequence CF_3 may come after that **par** operator. Parallel execution of two sequences meant that the full system state was the combination of the states of two parts, with the final number of states being the product of the number of states of the parts. Our MSRETG formally specified how this combination was applied to each type of combined fragment. Our MSRETG also allowed for a par symbol to allow the MSRETGs of the two parts to be written independently, in a more compact and readable way. The related work papers also contained a **par** operator. However, the complex semantics of Composition were not formally specified.

Table 2 summarizes the feature differences in terms of

Table 2. Feature differences in terms of semantics and syntax among the proposed SBC method and the other methods

Item	SBC	Other methods
Formal syntax and semantics for sequential behavior	The MSRETG provided for the loop is a glue for connecting the various sentences that define the system.	Few have given explicit grammar. There is confusion within the UML 2.0 community as to the proper representation of a sequence diagram.
Formal syntax and semantics for loops	The solution has been able to seamlessly compose across loops.	Loops for only for the trivial, bounded case
Formal syntax and semantics for alternative behavior	The grammar is clear that no sequence CF_3 may come after that alt operator.	The key limitation of other works is in Composition, particularly across loops.
Formal semantics for parallel behavior	MSRETGs of the two parts to be written independently, in a more compact and readable way.	The complex semantics of Composition was not formally specified.

semantics and syntax among the proposed SBC method and the other methods.

V. SUMMARY AND FUTURE WORK

A. Summary

This paper presents the SBC method for the formal specification of a sequence diagram as a MSRETG. The MSRETG is an example of a LTS that unifies structural and behavioral constructs. In order to formalize the creation of the MSRETG, the semantic meaning of SD modules was presented, including the meaning of these modules when they were used in a hierarchy that defined another module. Such hierarchical decomposition was formally provided for all sequence diagram modules, including all types of loops. The obtained MSRETG has advantages over previous formal specifications because it is a fully-complete specification and compact and readable.

A trace refers to any possible observed event sequence that could be observed in the operation of a system. The set of all possible traces is a formal description of a system. The LTS specification contained in the set of all paths through the graph (in our case, MSRETG) is equivalent to the set of all traces of the system. The advantages of the LTS description include its more compact description and its more-intuitive representation of the system, which makes it more useful for communicating and debugging system behavior.

B. Future Work

The MSRETG is an example of a label transition system that follows the concept of SBC. Therefore, MSRETG provides a compact and intuitive semantic description. This description is useful for providing an alternative view of a sequence diagram. This can be helpful for visual debugging, as well as for automated analysis of illegal conditions or of consistency checking with other UML diagram types. This could be an area of future investigation.

It is also possible to extend the representation. Although not discussed, negative conditions and real-time constraints could be added to the specification. The formal syntax and semantics of the approach would facilitate such additions.

ACKNOWLEDGMENTS

The authors are thankful to the anonymous reviewers for their useful remarks, which helped to clarify ambiguities and generate new ideas.

REFERENCES

1. M. Blaha and J. Rumbaugh, *Object-Oriented Modeling and Design with UML*, 2nd ed. Upper Saddle River, NJ: Pearson Education, 2005.
2. G. Booch, *Unified Modeling Language User Guide*. Boston, MA: Addison-Wesley Professional, 2017.
3. Object Management Group, "OMG Unified Modeling Language (version 2.3)," 2010; https://www.omg.org/spec/UML/2.3/About-UML/.
4. J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Upper Saddle River, NJ: Addison-Wesley, 2005.
5. O. Haugen, K. E. Husa, R. K. Runde, and K. Stolen, "STAIRS towards formal design with sequence diagrams," *Software & Systems Modeling*, vol. 4, no. 4, pp. 355-357, 2005.
6. Y. Hammal, "Branching time semantics for UML 2.0 sequence diagrams," in *Formal Techniques for Networked and Distributed Systems – FORTE 2006*. Heidelberg, Germany: Springer, 2006, pp. 259-274.
7. M. V. Cengarle and A. Knapp, "An institution for UML 2.0 interactions," 2008; http://mediatum.ub.tum.de/doc/1094623/TUM-I0808.pdf.
8. A. Knapp and J. Wuttke, "Model checking of UML 2.0 interactions," in *Model Driven Engineering*. Heidelberg, Germany: Springer, 2006, pp. 42-51.
9. M. V. Cengarle, P. Graubmann, and S. Wagner, "Semantics of UML 2.0 interactions with variabilities," *Electronic Notes in Theoretical Computer Science*, vol. 160, pp. 141-155, 2006.
10. H. Dan, R. M. Hierons, and S. Counsell, "A thread-tag based semantics for sequence diagrams," in *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, London, UK, 2007, pp. 173-182.
11. B. V. Selic, "On the semantic foundations of standard UML 2.0," in *Formal Methods for the Design of Real-Time Systems*. Heidelberg, Germany: Springer, 2004, pp. 181-199.
12. M. V. Cengarle, A. Knapp, A. Tarlecki, and M. Wirsing, "A heterogeneous approach to UML semantics," in *Concurrency, Graphs and Models*. Heidelberg, Germany: Springer, 2008, pp. 383-402.
13. K. P. Lin and W. S. Chao, "The structure-behavior coalescence approach for systems modeling," *IEEE Access*, vol. 7, pp. 8609-8620, 2019.
14. W. M. Ma and W. S. Chao, "Structure-behavior coalescence abstract state machine for metamodel-based language in model-driven engineering," *IEEE Systems Journal*, vol. 15, no. 3, pp. 4105-4115, 2021.
15. R. Milner, *Communication and Concurrency*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
16. R. Milner, *Communicating and Mobile Systems: The Pi Calculus*. Cambridge, UK: Cambridge University Press, 1999.
17. J. A. Bergstra and J. W. Klop, "ACP: a universal axiom system for process specification," *CWI Newsletter*, vol. 15, pp. 3-23, 1987.
18. W. S. Chao, *Single-Queue SBC Process Algebra for Systems Architecture: The Structure-Behavior Coalescence Approach*. Scotts Valley, CA: CreateSpace Publishing, 2015.

19. J. A. Goguen and R. M. Burstall, "Institutions: abstract model theory for specification and programming," *Journal of the ACM*, vol. 39, no. 1, pp. 95-146, 1992.

Steve Haga

Steve Haga received his M.E. and Ph.D. in electrical engineering from the University of Mayland, College Park, USA, in 1999 and 2005. He is currently an assistant professor of Computer Science and Engineering at National Sun Yat-sen University in Taiwan, ROC. His research interests include compilers, GPUs, and system modeling.

Wei-Ming Ma http://orcid.org/0000-0002-9334-1415

Wei-Ming Ma received his B.E. degree in navigation-engineering from the Chinese Naval Academy, Taiwan, ROC, in 1981, an M.S. degree in hydrographic sciences from the Naval Postgraduate School, CA, USA, in 1988, and a Ph.D. (1997) in physical oceanography from the Florida Institute of Technology, FL, the USA in 1997. He was an assistant professor in the Department of Information Management at Cheng-Shiu University, Taiwan, ROC, from 2002 to 2015. He has been an associate professor since Jan. 2016. His research interests include SBC architecture, enterprise architecture, information security, digital forensics, and AR/VR multimedia design.

William Shanchung Chao

William Shanchung Chao received his B.E. degree in communications engineering from National Chiao Yung University, Taiwan, ROC, in 1976, an M.E. and Ph.D. in information science from the University of Alabama at Birmingham, USA, in 1985 and 1988. He worked as an associate professor and a computer scientist at GE Research and Development Center from 1988 to 1991. He was an associate professor in the Department of Information Management at National Sun Yat-Sen University, Taiwan, ROC, since 1992. He is a member of the association of enterprise architects of Taiwan Chapter and is also a member of the Chinese Association of Enterprise Architects. His research interests include systems architecture, hardware architecture, software architecture, and enterprise architecture.