Formally Verifying Sequence Diagrams for Safety Critical Systems

Xiaohong Chen, Frédéric Mallet, Xiaoshan Liu

Formally Verifying Sequence Diagrams for Safety Critical Systems

Xiaohong Chen*, Frédéric Mallet †, Xiaoshan Liu *
*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, 200062, China
†Université Côte d'Azur, CNRS, Inria, I3S, France

***Abstract*—UML interactions, aka sequence diagrams, are frequently used by engineers to describe expected scenarios of good or bad behaviors of systems under design, as they provide allegedly a simple enough syntax to express a quite large variety of behaviors. This paper uses them to express formal safety requirements for safety critical systems in an incremental way, where the scenarios are progressively refined after checking the consistency of the requirements. As before, the semantics of these scenarios are expressed by transforming them into an intermediate semantic model amenable to formal verification. We rely on the Clock Constraint Specification Language (CCSL) as the intermediate semantic language. An SMT-based analysis tool called MyCCSL is used to check consistency of the sequence diagrams. We compare these requirements against actual execution traces to prove the validity of our transformation. In some sense, sequence diagrams and CCSL constraints both express a family of acceptable infinite traces that must include the behaviors given by the finite set of finite execution traces against which we validate. Finally, the whole process is illustrated on partial requirements for a railway transit system.**

***Index Terms*—Safety Critical Systems; Sequence Diagram; Clock Constraint Specification Language; Formal Verification; Safety Requirements**

I. Introduction

The elicitation and formalization of safety requirements are very important steps in today's development processes of safety critical systems, especially if they have to follow safety critical software development standards such as railway application safety standards EN50129 [1]. In these standards, formal methods are strongly recommended to ensure safety because they could provide precise analysis.

But formal languages are not easy to learn and use. As a trade-off, engineers start to use semi-formal language such as UML (Unified Modeling Language) to specify requirements. As a kind of behavioral diagram, UML sequence diagram provide allegedly a simple enough syntax to express a quite large variety of behaviors, including expected scenarios of good or bad behaviors of systems under design. So it is natural to use sequence diagrams to describe safety requirements. By safety requirements, we mean requirements that describe what actions and/or constraints should or should not be performed to maintain the system in a safe state. Different from other requirements, safety requirements not only describe what the system should do, but also it should not do [2].

However, sequenced diagrams cannot be automatically analyzed and verified. There are many researches formalizing them. They range from early PROMELA-based model [3], timed automata network [4], calculus of communicating system [5] to various petri nets, including normal petri nets [6], coloured Petri nets [7], Queueing Petri Nets [8], and Generalized Stochastic Petri Nets [9]. However, they do not deal with negative operator which is necessary for safety requirements. Moreover, as requirements are usually provided in an incremental way, where the scenarios are progressively refined, consistency checking is essential. As some of their requirements are expressed in LTL or TCTL, it is difficult to check the consistency of requirements.

In this paper, we propose to extend the sequence diagrams, and to provide an automated consistency verification for them. As in many approaches before, the semantics of these scenarios are expressed by transforming them into an intermediate semantic model amenable to formal verification. We rely on the Clock Constraint Specification Language (CCSL) [10] as the intermediate semantic language. The CCSL as a companion language of MARTE (Modeling and Analysis Real-Time Embedded Systems) [11], a profile of UML, describes the causal and temporal relations among events in terms of logic clocks. It is direct to describe "after ...it should do" , or "it should not do..." in CCSL constraints.

Moreover, the CCSL constraints come up with analysis tools. For example, MyCCSL, an SMT-based CCSL constraints analysis tool [12]. It translates the CCSL constraints into SMT formula, and solves these constraints using the efficient SMT solver Z3 [13]. MyCCSL supports validity proving, trace analysis, dead lock detection, LTL model checking, and schedulability analysis. In this paper, we will use schedulability analysis to support consistency checking.

In order to prove the correctness of our transformation, we simulate the behaviors of sequence diagrams in CCSL semantics. It turns out, sequence diagrams and CCSL constraints both express a family of acceptable infinite traces that must include the behaviors given by the finite set of finite execution traces against which we validate. Finally, the whole process is illustrated on partial requirements for a railway transit system. To sum up, the contribution of this paper is to transform sequence diagrams to CCSL, and verify them against consistency using MyCCSL automatically, which makes sequence diagrams suitable for safety critical systems.

The remainder of the paper is organized as follows. Section II introduces CCSL and MyCCSL. Section III details the transformation from sequence diagrams to CCSL constraints. Section IV proves the correctness of our transformation, and

consistency checks CCSL constraints. Section V conducts a case study. Section VI compares related work, and finally Section VII concludes.

II. Preliminaries

This section introduces CCSL clock constraints and MyCCSL. CCSL provides descriptions of the causal and temporal relations among events in terms of logic clocks. The relations are constrained by clock constraints such as *strictPre*($<$). For better understanding, three other concepts, *clock*, *instant*, *instant relation* should be mentioned.

Each event could be a logical clock, which models the access to time. According to [14], a *clock* is defined as sequence of instants, i.e., $C :=< I, \prec >$ where, I is a (discrete) set of instants, $\prec$, named strict precedence is a quasi-order relation on I, where an *instant* is actually an occurrence of the event, which is also called a tick of clock. $C[k]$ denotes the k_{th} instant in I, and $k \in N_{>0}$, is called the index.

Instants in clocks have relations, including *precedence*, *strict precedence*, and *coincidence*. Precedence ($\preceq$) represents causal dependency relation between instants. Coincidence ($\equiv:=\preceq \cap \succeq$) is a strong relation that forces simultaneous occurrences of instants. Strict precedence ($\prec:=\preceq \setminus \equiv$) represents the sequential relation of occurrences of instants.

The clock constraints could be defined by millions of *instant constraints*. In this paper, we will use the following clock constraints, *strictPre*($<$), filterBy($\blacktriangledown$), exclude($\#$), alternate($\sim$), delayFor ($), subClock ($\subseteq$) and boundedDiff(-). Table I gives definitions of these clock constraints and their explanations.

MyCCSL is a tool for analyzing CCSL constraints. Its main function is to translate the CCSL constraints into SMT formula. The most effective SMT solver Z3 [13] is integrated in. As the SMT could effectively mitigate state explosion problem in model checking, MyCCSL is very effective. MyCCSL supports validity proving, trace analysis, deadlock detection, LTL model checking, and schedulability analysis [12]. In this paper, we will use schedulability analysis to support consistency checking of safety requirements. If the constraints are schedulable, MyCCSL returns *sat* and a schedule. Else, *unsat* is returned with a counterexample.

III. Mapping from Sequence diagram to CCSL

As the definitions and semantics of UML 2 sequence diagrams are rather informal and ambiguous [15], we facilitate the automated creation of the CCSL constraints by defining transformation rules for formally representing constructs of sequence diagrams. The main objective is to represent the sequence diagram's constructs and their relationships in a CCSL formalism. It facilitates the consistency verification.

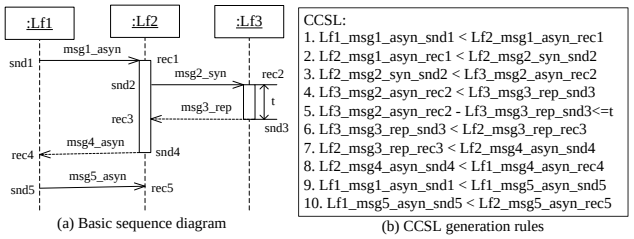
In the sequence diagram, there are many object lifelines. A lifeline has many interactions. Interactions are messages along with sending and receiving OccurrenceSpecifications (OSs) covered by lifelines in temporal order, associated CombinedFragments included operands. An interaction e is called "receiving event" of interaction i "sending event" if there is a link from i to e that synchronizes with the appropriate sending and receiving lifelines.

In our formalization, we view a message and its sending and receiving OSs (i.e., events) as a tuple $< Lf, msg, snd/rec >$, where Lf represents the lifeline that is responsible for sending or receiving messages, msg denotes the message name, snd and rec represent the sending OS and receiving OS of the corresponding message on a lifeline, respectively. We map this tuple into **two clocks**, snd, and rec. In order to be more precisely describe the snd, and rec, we name them $Lf - msg - snd$, and $Lf - msg - rec$.

For each sequence diagram mapping, we first find all the lifelines, the sending and receiving events, and finally find the causal and temporal relations among these clocks. According to [15], transformation rules are made by considering the following situations. Due to limited space, the mapping rules for Option, Break, Ignore and Consider combined fragments into CCSL are omitted.

A. Basic Sequence Diagram

A sequence diagram without a CombinedFragment is referred to as a basic sequence diagram (such as the one in Figure 1(a)). The following rules for sending and receiving messages must be considered as the semantics of a basic sequence diagram suggests.



(a) Basic sequence diagram

(b) CCSL generation rules

Fig. 1. Transformation of basic sequence diagram

- The OSs on the same lifeline must occur in the same order in which they are described [8, p.505], which means they keep the *strictPre* relations on the same lifeline.
- "The semantics of a complete message is simply the trace sendEvent, receiveEvent" [8, p.507]. A receiving OS of a message is enabled for execution if and only if the sending occurrence of the same message has already occurred [8, p.507]. This means these events have the *strictPre* relation: $sendEvent < receiveEvent$.
- If sending and receiving OSs of the same message are on the same lifeline then the sending event of a message must exist before its receiving event [8, p.506]. This also keeps the *strictPre* relations between them.
- For the synchronized message *syn_msg*, only after the response *msg_rep*, the next message *nmsg* could be sent. So we get: $syn\_msg\_snd < msg\_rep$ and $msg\_rep < nmsg\_snd$.
- For the asynchronized message, *asyn_msg*, no matter whether response *msg_rep* is received or not, the next

TABLE I
CLOCK CONSTRAINTS USED IN THIS PAPER

Clock constraint	semantic description	description
C_1 *strictPre* C_2 ($C_1 < C_2$)	$\forall i \in N^*, C_1[i] \prec C_2[i]$	restricts that the i_{th} instant of C_1 ticks strictly before the i_{th} instant of C_2.
C_1 *alternate* C_2 ($C_1 \sim C_2$)	$\forall i \in N^*, C_1[i] \prec C_2[i] \wedge C_2[i] \prec C_1[i+1]$	restricts an alternation of ticks of the left and right clocks.
$C_1 = C_2$ *union* C_3 ($C_1 = C_2 + C_3$)	$\forall i \in N^*, \exists j,k \in N^*, (C_1[i] \equiv C_2[j]) \vee (C_1[i] \equiv C_3[k])$	defines a clock C_1 such that C_1 ticks iff C_2 or C_3 ticks
C_1 *subClock* C_2 ($C_1 \subseteq C_2$)	$\forall i \in N^*, \exists j \in N^*, C_1[i] \rightarrow C_2[j] \equiv C_1[i]$	expresses that C_1 occurs at some step only if C_2 occur at this step as well
C_1 *exclude* C_2 ($C_1 \# C_2$)	$\forall i,j \in N^*, \neg C_1[i] \equiv C_2[j]$	restricts that the two clocks never tick at the same time.
C_1 *boundedDiff_i_j* C_2 ($i \leq C_1 - C_2 \leq j$)	$\forall k \in N^*, C_1[k+i] \preceq C_2[k] \preceq C_1[k+j+1]$	restricts that the bounded drifts between the two logical clocks must be within [i,j].
$C = C_1$ filteredBy w ($C = C_1 \blacktriangledown w$)	$\forall k \in N^*, C[k] \equiv C_1[w \uparrow k]$ where $w \uparrow k$ represents the kth 1 in w	creates a subclock C of C_1 according to the binary word w.
$C_1 = C_2$ delayFor d on C_3 ($C_1 = C_2$ \$ d on C_3)	$\forall m \in N^*, \exists i,j,k \in N^*, (C_1[i] \equiv C_2[k]) \wedge (C_3[j] - C_3[m] = d)$	when C_3 ticks d times, C_1 ticks together with C_2

message *nmsg* could be sent, i.e., $syn\_msg\_snd < msg\_rep$, and $syn\_msg\_snd < nmsg\_snd$.

- For the time duration t between two events eve_1, eve_2 (no matter sending or receiving events of messages), there should be time constraints t between these two events: $0 =< eve_1 - eve_2 <= t$, where $0 =<$ would be omitted.

As the *strictePre* relation is transitive, there is not necessary to write down all of them. In searching the sequence diagram, we use the deep first algorithm to find all the messages, and translate them. Figure 1(a) shows an example of basic sequence diagram. *Msg1_asyn* is a message sent from lifeline *Lf1* to the lifeline *Lf2*. The particular sent/receiving rule is shown in Figure 1(b) line 1. *Msg2_syn* is a synchronized message, with *msg3_rep* being its response. There is a duration t between the receiving *msg2_syn* and sending *msg3_rep*. Only after receiving *OS* (rec3) is enabled, the next message *msg4_asyn* could occur. These rules are Figure 1(b) line 4-8. *Msg1_asyn* is also an asynchronized message, with *msg4_asyn_rec* being its response, and *msg5_asyn* being its next message. The rule in shown in Figure 1(b) line 9-10.

B. Weak Sequencing Combined Fragment

The **seq** interaction operator imposes the order of the execution of operands associated with the same lifeline with the following constraints [8, p.483]:

- The ordering of events (i.e., OSs) within each of the operands are maintained.
- OSs on different lifelines from different operands may execute in any order.
- OSs on the same lifeline from different operands are ordered such that an OS of the first operand comes before that of the second operand.

Figure 2(a) shows an example of a Weak Sequencing combined fragment with the same lifelines from different operands. Figure 2(b) illustrates the rules for mapping a Weak Sequencing combined fragment into CCSL constraints. In operand 1, message $msg1$ and $msg2$ keep the same rules as in the basic sequence diagram as shown in line 1-4 of Figure 2(b). In operand 2, message $msg3$ share the same lifelines with $msg1$ and $msg2$. For $Lf1$, the OS $snd2$ must happen

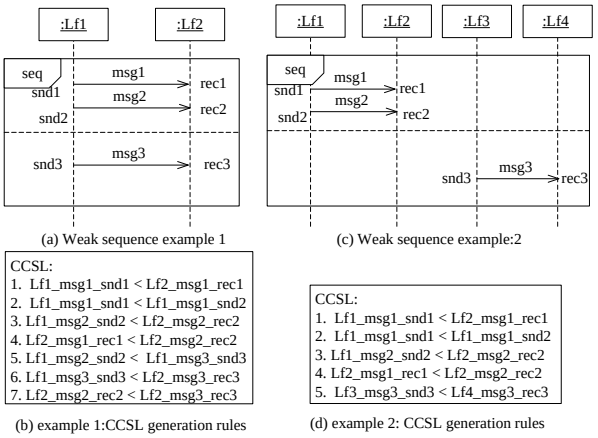


Fig. 2. Transforming of Weak Sequencing combined fragment

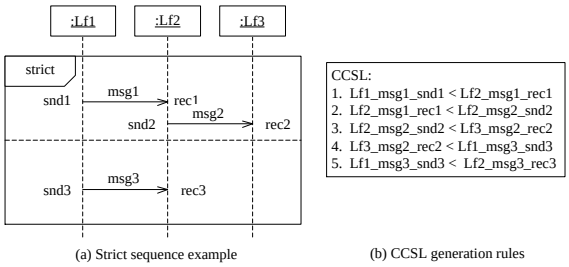
before $snd3$, while for $Lf2$, the OS $rec2$ must happen before $rec3$. So we get rules of $msg3$ in line 5-7 in Figure 2(b).

Figure 2(c) is another example of a Weak Sequencing combined fragment with different lifelines from different operands. Figure 2(d) illustrates the rules for mapping a Weak Sequencing combined fragment into CCSL constraints. In operand 1, message $msg1$ and $msg2$ keep the same rules as in the basic sequence diagram as shown in line 1-4 of Figure 2(d). In operand 2, $msg3$ have different lifelines with $msg1$ and $msg2$. The occurrence of $msg3$ is independent from $msg1$ and $msg2$. So we only get a rule in line 5 in Figure 2(d).

C. Strict Sequencing Combined Fragment

The semantics of **Strict** Sequencing (i.e., strict interaction operator) imposes the total order between adjacent operands. It contains a stronger version of the second rule introduced for Weak Sequencing, in particular, OSs on different lifelines from different operands have strict order of execution [8, p.483]. In other words, the first OS in a succeeding operand cannot be enabled until all the OSs on all the covered lifelines within the preceding operand have completed. Any covered lifeline needs to wait for other lifelines to enter the second or subsequent operand. For instance, sending OS of a message $msg3$ within the second operand covered by a lifeline $Lf1$ will not be executed until the last OS that is $rec2$ within a first

operand on a lifeline $Lf3$ finishes its execution, as shown in Figure 3(b) (Line 1-4). Figure 3(b) shows its CCSL generation rules.



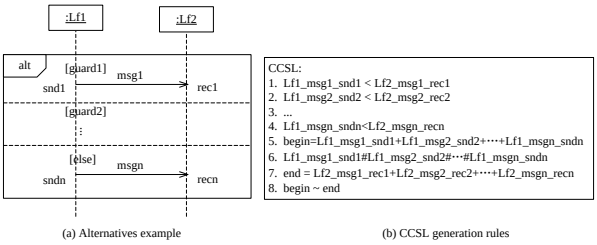
(a) Strict sequence example (b) CCSL generation rules

Fig. 3. Transformation of Strict Sequencing combined fragment

D. Alternatives

In the UML 2 specification [8, p.482], an Alternative combined fragment describes a branching operation in a sequence diagram. The **alt** operator of the combined fragment represents a choice of behavior where at most one of the operands will be selected whose interaction constraint (guard condition) evaluates to True (i.e., an if-then-else statement). The else guard is the negation of the disjunction of all other constraints in the enclosing combined fragment. If none of the operands has a guard that evaluates to True, none of the operands will be executed and the remainder of the enclosing InteractionFragment will be performed. This also means, only one of the OS in these operands occurs. They are exclusive.

Figure 4(a) shows an example of an Alternative combined fragment, whose guard is encoded as a boolean variable. If the guard of the first operand evaluates to True, the OSs enclosed within the first operand are executed, otherwise the whole operand is skipped. In this operand, the messages hold the temporal relations (line 1-4 in Figure 4(b)). The choice among alternatives is made using *union* and *exclude* relations in Figure 4(b) (Line 5-6). To ensure the corresponding rec of snd is certainly to happen, we union all the $recs$ and use *alternate* as shown in Figure 4(b) (Line 7-8).



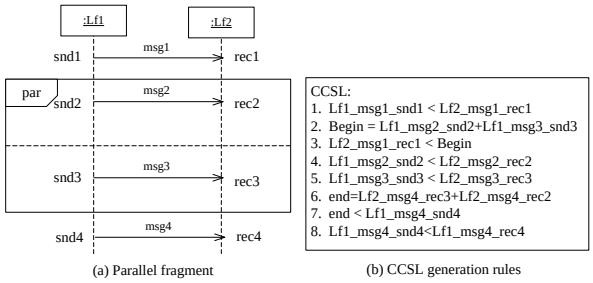
(a) Alternatives example (b) CCSL generation rules

Fig. 4. Transformation of Alternative combined fragment

E. Parallel Combined Fragment

A Parallel combined fragment is denoted by an interaction operator **par** which defines potentially parallel merge execution of behaviors of the operands [8, p.483]. The OSs of different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved. In other words, OSs of messages within the same operand respect the order along a lifeline whilst OSs of messages on the same lifeline from different operands are ordered such that the first message occurrence of the operands has the same preceding OS.

Figure 5(a) shows an example of parallel combined fragment, and Figure 5(b) shows its transformation into CCSL constraints. In this figure, a sending OS of a message $msg1$ ($Lf1\_msg1\_snd1$) on a lifeline $Lf1$ leads to the execution of sending OSs of messages in both operands (i.e., $Lf1\_msg2\_snd2$ and $Lf1\_msg3\_snd3$). It is translated to line 1-3. Similarly, the receiving OS of messages in both operands, (i.e., $Lf2\_msg2\_rec2$ and $Lf2\_msg3\_rec3$), trigger $msg4$, transformed into constraints Line 6-8 in Figure 5(b).



(a) Parallel fragment (b) CCSL generation rules

Fig. 5. Transformation of Parallel combined fragment

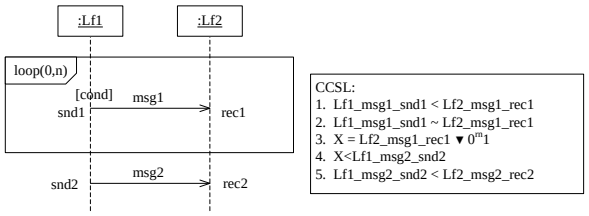
F. Loop Combined Fragment

In Loop combined fragments, the interaction operator **loop** defines that its sole operand will be repeated for at-least the minimum (minint) number of times and at-most maximum (maxint) number of times as long as the guard condition remains True [8, p.485]. If the loop has no bounds, this means that an indefinite loop (with minint = 0 and maxint = infinite) is executed. However, it is unrealistic for most loops that they really execute indefinitely, and therefore, we assume that loops will eventually stop.

Figure 6(a) shows an example of a Loop combined fragment having minint = 0 and maxint = n. To deal with Loop combined fragments, we assume the execute number is counter. Set it to 0 at first. After the end of the current iteration, the counter is increased by one at the beginning of the next iteration. Furthermore, the loop condition and counter are checked at the beginning of each iteration. If the condition is evaluated to false or counter is greater or equals to maxint, a new iteration cannot start and execution of the loop will terminate. The OSs of all the messages within the operand among iterations execute sequentially along a lifeline. Suppose the real execution number is rn, where $0 \leq rn \leq n$, we bring in a temporary variable clock X for filtering rn_{th} occurrence of $Lf2\_msg1\_rec1$ (Line 3). X will certainly happen before the sending OS ($snd2$), which is shown in line 4 Figure 6(b).

G. Negative Combined Fragment

The interaction Operator **neg** designates that the Combined-Fragment represents traces that are defined to be invalid [8,

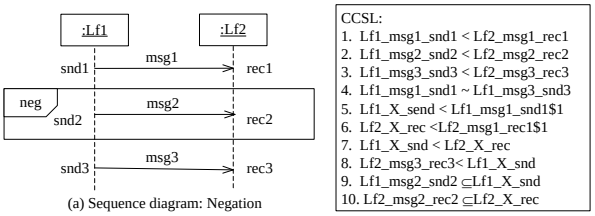


(a) Loop fragment (b) CCSL generation rules

Fig. 6. Transformation of Loop combined fragment

p.482]. The set of traces that defined by operator negative is equal to the set of traces given by its (sole) operand, only that this set is a set of invalid rather than valid traces. In another word, the negation operator is used to denote the case of not-executing some events, or forbidden behaviour. Therefore, it is particularly suitable for safety requirements.

Figure 7(a) shows an example of negative combined fragment, and 7(b) shows the corresponding CCSL constraints. In this figure, a sending OS of a message $msg2$ ($Lf1\_msg2\_snd2$) on the lifeline $Lf1$ is forbidden to happen between $msg1$ and $msg3$. So it is only allowed to occur between $msg3$ and next time $msg1$. We create a virtual message X. It happens after $msg3$, but before next $smg1$ (Line 5-8 in Figure 7(b)). $msg2\_snd$ is a subclock of X_snd, and $msg2\_rec$ is a subclock of $X.rec$ (Line 9-10 in Figure 7(b)). This makes 'allow' which means it may happen or not.



(a) Sequence diagram: Negation (b) CCSL generation rules

Fig. 7. Transformation of Negative fragment

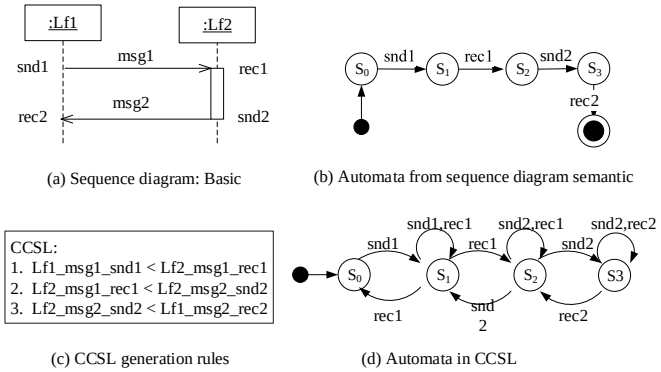
IV. Proof of transformation and consistency checking of safety requirements

A. Proof of transformation

In this section, we analyse each typical sequence diagram, and try to find the relation between the sequence diagram semantic and CCSL constraint semantic.

For basic sequence diagram, weak sequencing combined fragment, and strict sequencing combined fragment, in fact they are of *strictPre* relation. So we only consider two messages in a sequence diagram as shown in Figure 8(a). From the semantics of sequence diagram, we could get an FSM as shown in Figure 8(b). It means, the trace is, $< snd1, rec1, snd2, rec2 >$. The corresponding CCSL constraints are shown in Figure 8(c). Reviewing it in the strictPre semantic, we only use n=1 to bound it. Then the automata is shown in Figure 8(d). From the two FSMs, we can see that, (d) could simulate (b), but (b) could not simulate (d).

For the alternatives, we use the most simple situation (Figure 9(a)). Only two messages are given. From the sequence

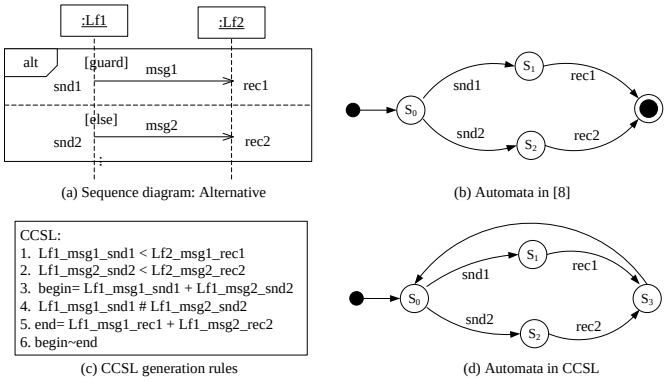


(a) Sequence diagram: Basic (b) Automata from sequence diagram semantic

(c) CCSL generation rules (d) Automata in CCSL

Fig. 8. Sequence diagram of strictPre

diagram semantics, an FSM is obtained in Figure 9(b). Either $snd1$, $rec1$ happens, or $snd2$, $rec2$ happens. The corresponding CCSL constraints are shown in Figure 9(c). According to *strictPre*, *union*, and *exclude* semantics, the corresponding FSM of these constraints are given in Figure 9(d). Obviously (d) could simulate (b), but not the other way around.



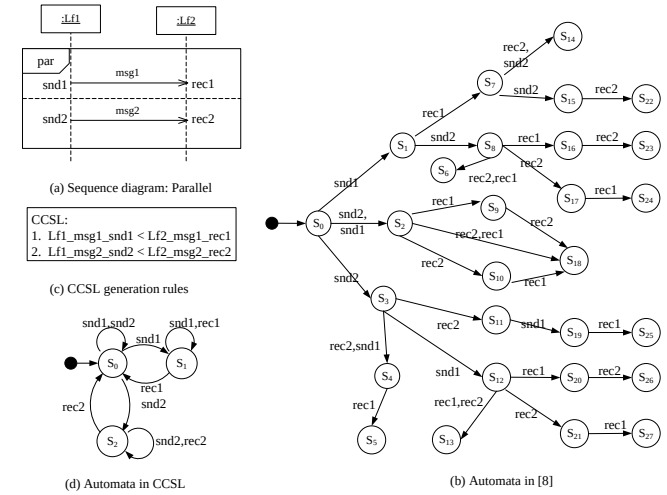
(a) Sequence diagram: Alternative (b) Automata in [8]

(c) CCSL generation rules (d) Automata in CCSL

Fig. 9. Sequence diagram of Alternate

For the parallel combined fragment, we also choose only two operands with two messages in Figure 10(a). According to the sequence diagram semantics, $snd1$, $snd2$, $rec1$, $rec2$, could happen in any order only $snd1$ ($snd2$) must be before $rec1$ ($rec2$). So we get a FSM in Figure 10(b). In CCSL, they are two *strictPres* (Figure 10(c)). We compose the two semantics together, then get (d).

For the loop combined fragment, we also choose two loops in Figure 11(a). According to the sequence diagram semantics, $< snd1, rec1 >$ happens two times, $snd2$, $rec2$, could happen once. So we get the FSM in Figure 11(b). In CCSL, they are compositions of *alternate*, *strictPre*, and *filteredBy* (Figure 11(c)). We compose the three semantics together, then get (d). To sum up, in each figure (from Figure 8 to Figure 11), the automata in (d) could simulate (b), which means our semantics have more state pace. This shows that our transformation is not an equal. But when doing the verification, if the safety requirements pass our consistency verification, it will surely



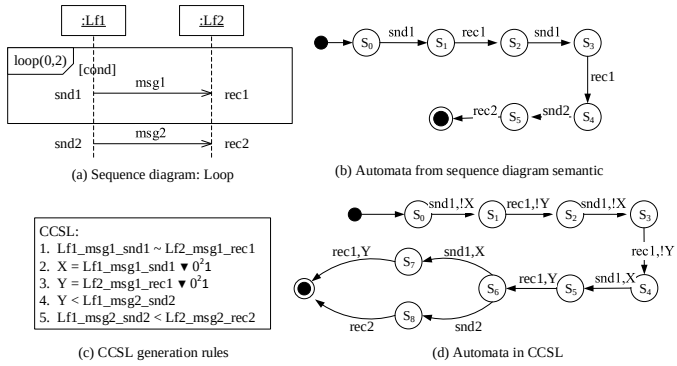
(a) Sequence diagram: Parallel

(c) CCSL generation rules

(d) Automata in CCSL

(b) Automata in [8]

Fig. 10. Sequence diagram of Parallel



(a) Sequence diagram: Loop

(b) Automata from sequence diagram semantic

(c) CCSL generation rules

(d) Automata in CCSL

Fig. 11. Sequence diagram of Loop

satisfy the consistency requirements of sequence diagrams.

B. Consistency checking of safety requirements

In MyCCSL, the schedulability analysis is used to find whether there is clock ticks. In our consistency checking, if the safety requirements are inconsistent, there must be no clock ticks. Therefore, we only need to check whether the safety requirements in terms of events are schedulable or not.

As a model checker, MyCCSL only supports the core CCSL expressions. Not all the clock constraints used in this paper are defined in MyCCSL. Constraints *alternate* and *boundedDiff* need to be transformed. The *alternate* ($C_1 \sim C_2$), could be expressed using the *delayFor*(\$). Between $C_1[i]$ and next $C_1[i+1]$, $C_2[i]$ could be inserted. The *boudedDiff* (-)could be described using the *delayFor* and *idealClock*, where the *idealClock* is the watch time in the MyCCSL. Each tick of *idealClock* can be 1 millisecond.

Before the verification, one needs to input the running parameter: the program timeout period to prevent the running time from being too long to terminate; the SMT solver can choose Z3 to solve the SMT formulae; the boundary value, 0 represents unbounded verification, usually we will enter a positive integer as the bound of verification; and whether this is periodic or not. Then one can click *Run*. It will give you a result *sat* or *unsat* in a short time.

V. CASE STUDY

This section uses a realistic scenario, Automatic Operation System Platform Departure scenario, which is motivated by a railway accident happened at 6:16 pm on the 5th July of 2010. It happened at Zhongshan Park Station of Shanghai Rail Transit Line 2. When the train was closing, a passenger wanted to force the door, but she got caught on the wrist. After the train started, she was dragged, fell on the platform and died.

Figure 12 (without red lines) shows the related scenario, closing the train door, where TIAS represents a traffic integrated automation system; VOBC represents a vehicle on board: CI represents a computer interlocking system. The main interactions are as follows.

(1) TIAS sends close door command to VOBC, (2) VOBC sends close door comamnd to DoorController, (3) Door controller closes the door, and returns door status to VOBC, (4) VOBC sends train door status to CI, (5) CI send close Station door command to station door controller, (6) station door controller close station door and return the station door status to CI, (7) CI report station door status to VOBC, (8) CI report station door status to TIAS, (9) VOBC sends pull to train, (10) train starts to move, (11) VOBC sends stop command to train, and (12) the train ends move.

Figure 12 (without red lines) is a basic sequence diagram. We can directly translate it to CCSL constraints using rules in Figure 1(b). The CCSL constraints generated are listed in $https://github.com/Safety-req/TIAS$. Only for this scenario, there is no inconsistency. Our SMT based verification also proves this. We use MyCCSL tool to verify them. The verification is carried out on the following machine configurations, CPU Inter(R) Core(TM)i7-4790 @3.60GHz, 12.0 G RAM, and Windows 10 (64 bit). On MyCCSL, we choose the Z3 solver, and set bound 70, timeout 10000s. The verification result is 'sat', which means consistency after 912.78s.

After the accident, one may come up with a refined scenario which add new interactions as the red color messages in Figure 12 including: (1) When the train is moving, VOBC is forbid to open the train door. (2) If the response time of the train door is over 5 seconds, it should make an emergency brake. These two modifications would as the following constrains.

(1) As moving is a state, we translate it to two events, startMove, and endMove. Event open should happen after endMove, but before the next startMove. The constraints could be expressed following rules in Figure 7(omit lifeline):

$$openDoor\_snd < openDoor\_rec;$$
$$startMove\_snd < startMove\_rec;$$
$$stop\_snd < stop\_rec;$$
$$startMove\_rec \sim stop\_snd;$$
$$X < startMove\_snd \ \$ \ 1;$$
$$Y < startMove\_rec \ \$ \ 1 \ ;$$
$$X < Y;\ stop\_rec < X;$$
$$openDoor\_snd \subseteq X;$$
$$openDoor\_rec \subseteq Y;$$

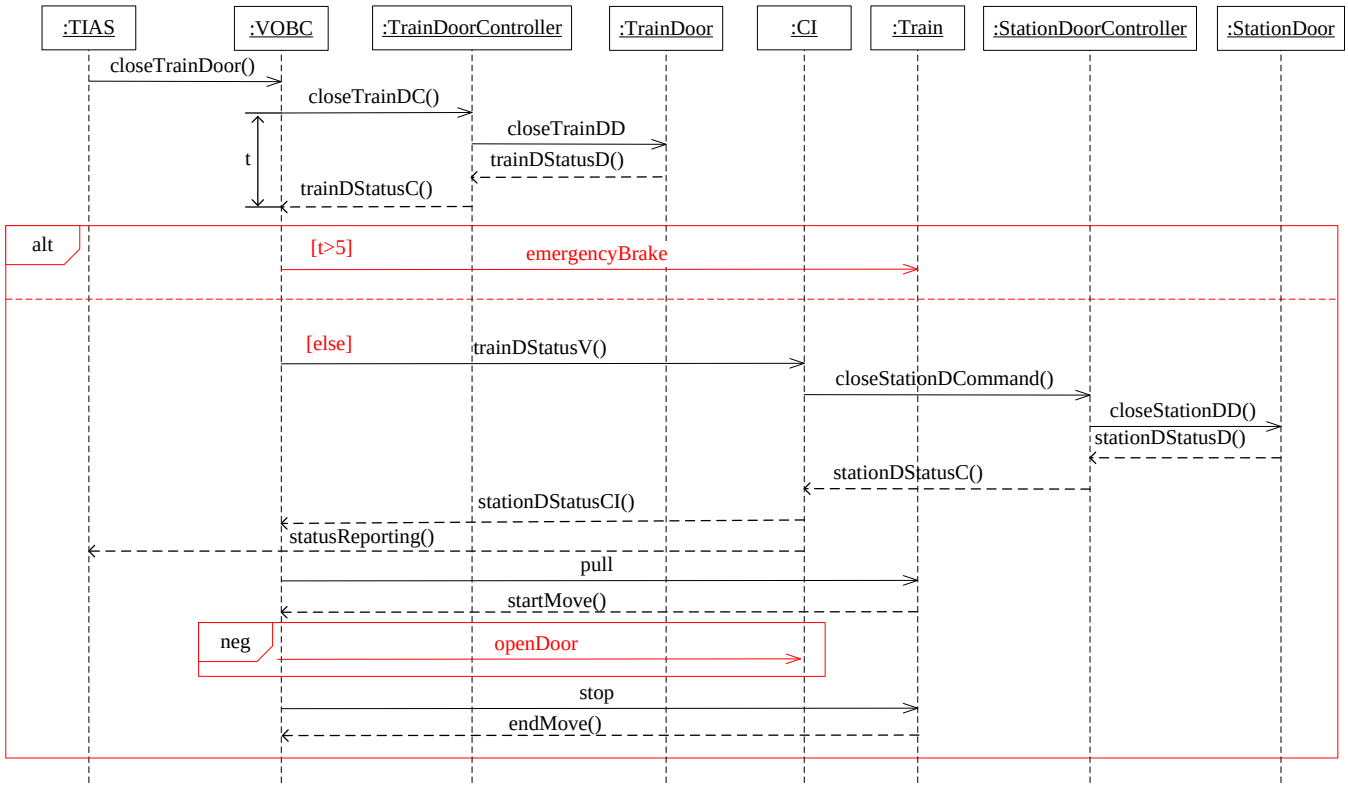


Fig. 12. Sequence diagram of the railway station scenario

(2) The response time of the train door is measured by the duration between VOBC sending a message 'closeTrainDC' and receiving a message 'trainDStatus'. According to the rules in Figure 4, we have (omit lifeline due to space):

$closeTrainDC\_snd - trainDStatusC\_rec <= 5000$;
$trainDStatusC\_rec < begin$;
$begin = trainDStatusV\_snd + emergencyBrake\_snd$;
$trainDStatusV\_snd \ \# \ emergencyBrake\_snd$;
$end = emergecyBrake\_rec + endMove\_rec$;
$begin \sim end$;
$Z = closeTrainDoorC\_snd \ \$ \ 5000 \ on \ idealClock$;
$Z < emergecyBrake\_snd$;

After modification, one puts all these constraints with the original constraints into MyCCSL. After with the same machine configuration, we get the result 'sat' which means consistent after 47.85s.

VI. Related work

There are many researches on formalizing sequence diagram 2.0 in various formalism. For example, Lima et al. [3] creates a PROMELA-based model from UML interactions expressed in sequence diagrams, and use SPIN model checker to simulate the execution and to verify properties written in Linear Temporal Logic (LTL). Han et al. [4] realize a full-map from sequence diagram to timed automata network. Formal verification is carried out using automated model checkers like UPPAAL. Tu et al. [5] establish formal specification rules for UML 2.0 diagrams based on CCS, calculus of communicating system. Theorems are developed to obtain the formal specification of UML 2.0 sequence diagrams with loop, alternative flow and concurrency. Li et al. [16] define a static semantics for UML sequence diagrams to support checking the well-formedness of an sequence diagram in the context of other diagrams, i.e. its consistency with a class diagram and a state diagram. Shen et al. [17] present a template semantics, for describing the operational semantics of behavioral notations, especially the combined fragments and other constructs of sequence diagrams. Muram et al. [18] translate sequence diagrams into temporal logic based constraints (LTL) and formal behavior specifications (i.e., symbolic model language (SMV)), respectively. The model checker NuSMV is used to

verify the containment relationship.

Various petri nets are used. For example, [6] defines and explains the relationship between sequence diagrams and normal Petri nets. Cunha et al. [19] also translate UML sequence diagrams to Petri nets and verify deadlock freeness, reachability, safety and liveness properties by using a model checker. Soares et al. [7] present an approach to automatically translate Sequence Diagrams to coloured Petri nets(CPN) ready for execution with CPN Tools. Bowles et al. [20] also transform UML 2 sequence diagrams to CPNs. Formal transformation rules are extended to consider modelling aspects such as stochastic and real-time behaviour. Existing Petri net analysis and verification tools are exploited. [8] transforms sequence diagrams into equivalent Queuing Petri Nets. Paper [9] proposes an automatic translation of Statecharts and Sequence Diagrams into Generalized Stochastic Petri Nets, and a composition of the resulting net models.

Among these works, firstly, they do not deal with negative operator which is important for safety requirements while our approach address that. Secondly, petri net is mostly used. But petri net has too strong expressiveness to be verified. Our CCSL when transformed to a causality clock graph is a simplified version of petri net [21]. Also CCSL provides various kinds of analysis tools such as safety analysis in [21] although in this paper we only focus on consistency analysis using the state-of-the-art model checker Z3. The paper is mostly related to Dhaou et al.'s work [22]. They have extended an existing causal semantics to deal with sequence diagrams with the most popular combined fragments (Alt, Opt and Loop), by processing the sequence diagram as a whole. They have proposed several rules to derive the partial orders between the events. In fact, CCSL constraints also describe event relations, and even provide more than partial orders.

VII. Concluding remarks

In this paper, we propose a formalization for safety requirements in terms of sequence diagrams. Our paper makes the following contributions: (1) The consistency property of safety requirements in terms of CCSL are checked by MyCCSL; (2) The formalization of safety requirements in terms of sequence diagram are realized by CCSL constraints; and (3) The correctness of our transformation from sequence diagrams to CCSL constraints is proved by simulation.

In the future, we want to use the counter examples returned from inconsistent cases to explain the reasons of inconsistency, and provide strategies to refine the safety requirements.

Acknowledgments

This research was supported by the National Key Research and Development Program of China under grant number 2018YFB2101300 and the French National Research Agency SIM ANR-19-CE25-0008-02.

References

[1] CENELEC, "50129: Railway application–Communications, signaling and processing systems–Safety related electronic systems for signaling," *European Committee for Electrotechnical Standardization*, 2018.

[2] N. G. Leveson, "Software safety in computer-controlled systems," *IEEE Computer*, vol. 17, no. 2, pp. 48–55, 1984.

[3] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi, "Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages," *Electr. Notes Theor. Comput. Sci.*, vol. 254, pp. 143–160, 2009.

[4] D. Han, J. Xing, Q. Yang, H. Wang, and X. Zhang, "Formal sequence: Extending UML sequence diagram for behavior description and formal verification," in *40th IEEE Annual Computer Software and Applications Conference, COMPSAC Workshops 2016, Atlanta, GA, USA, June 10-14, 2016*, 2016, pp. 474–481.

[5] T. Peng and G. Ding, "Formal specification and automated verification of uml2. 0 sequence diagrams," in *2012 IEEE International Conference on Granular Computing*, 2012, pp. 370–375.

[6] T. S. Staines, "Transforming uml sequence diagrams into petri net," *Journal of communication and computer*, vol. 10, no. 1, pp. 72–81, 2013.

[7] J. A. C. Soares, B. Lima, and J. P. Faria, "Automatic model transformation from UML sequence diagrams to coloured petri nets," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018*, 2018, pp. 668–679.

[8] V. V. Doc, H. Q. Thang, and N. T. Bach, "Development of the rules for transformation of UML sequence diagrams into queueing petri nets," in *Industrial Networks and Intelligent Systems - 4th EAI International Conference, INISCOM 2018, Da Nang, Vietnam, August 27-28, 2018, Proceedings*, 2018, pp. 122–144.

[9] S. Bernardi, S. Donatelli, and J. Merseguer, "From uml sequence diagrams and statecharts to analysable petri net models," in *Proceedings of the 3rd international workshop on Software and performance*, 2002, pp. 35–45.

[10] F. Mallet, "Clock constraint specification language: specifying clock constraints with UML/MARTE," *ISSE*, vol. 4, no. 3, pp. 309–314, 2008.

[11] OMG, "Uml profile for modelling and analysis of real- time and embedded systems (marte)," http://www.omgmarte.org/.

[12] M. Zhang, F. Dai, and F. Mallet, "Periodic scheduling for MARTE/CCSL: theory and practice," *Sci. Comput. Program.*, vol. 154, pp. 42–60, 2018.

[13] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, 2008, pp. 337–340.

[14] C. Andre, "Syntax and semantics of the clock constraint specification language (CCSL)," INRIA, Research Report, 2009.

[15] Object Management Group, *UML 2.4.1 superstructure specification*, http://www.omg.org/spec/UML/2.4.1.

[16] X. Li, Z. Liu, and J. He, "A formal semantics of UML sequence diagram," in *15th Australian Software Engineering Conference (ASWEC 2004), 13-16 April 2004, Melbourne, Australia*, 2004, pp. 168–177.

[17] H. Shen, A. Virani, and J. Niu, "Formalize UML 2 sequence diagrams," in *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008, Nanjing, China, December 3 - 5, 2008*, 2008, pp. 437–440.

[18] F. U. Muram, H. Tran, and U. Zdun, "A model checking based approach for containment checking of UML sequence diagrams," in *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, 2016, pp. 73–80. [Online]. Available: https://doi.org/10.1109/APSEC.2016.021

[19] E. Cunha, M. Custódio, H. Rocha, and R. S. Barreto, "Formal verification of UML sequence diagrams in the embedded systems context," in *Brazilian Symposium on Computing System Engineering, SBESC 2011, Florianopolis, Brazil, November 7-11, 2011*, 2011, pp. 39–45.

[20] J. Bowles and D. A. Meedeniya, "Formal transformation from sequence diagrams to coloured petri nets," in *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, 2010, pp. 216–225.

[21] F. Mallet, J. Millo, and R. de Simone, "Safe CCSL specifications and marked graphs," in *11th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMCODE 2013, Portland, OR, USA, October 18-20, 2013*, 2013, pp. 157–166.

[22] F. Dhaou, I. Mouakher, J. C. Attiogbé, and K. Bsaïes, "A causal semantics for UML2.0 sequence diagrams with nested combined fragments," in *ENASE 2017 - Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, Porto, Portugal, April 28-29, 2017*, 2017, pp. 47–56.