Information Engineering Express
International Institute of Applied Informatics
2020, Vol. 6, No. 2, 1 - 19

Synthesis and Consistency Verification of UML Sequence Diagrams with Hierarchical Structure

Akira Matsumoto [*], Tomoyuki Yokogawa [†],
Sousuke Amasaki [†], Hirohisa Aman [‡], Kazutami Arimoto [†]

Abstract

Automatic consistency checking for UML state machine diagrams and sequence diagrams has been expected since developers spend considerable effort to keep the consistency. In particular, the verification of diagrams with a hierarchical structure is required. Although formalization of state machine diagrams with hierarchical structure has been widely treated, it is not yet sufficient for sequence diagrams. In this paper, we propose an automatic method for verifying the consistency between state machine diagrams and sequence diagrams with a hierarchical structure. In our verification framework, the consistency of diagrams is defined as an inclusive relation between the sets of traces and is checked using the FDR model checker. FDR can check refinement relation between processes described as CSP_M notation. We provide a process description of state machine diagrams and sequence diagrams and can verify the consistency by checking traces inclusion of processes using FDR. Our description supports for sequence diagrams with a hierarchical structure. We applied the proposed process representation to an example diagram that describes interactions of basic components of a wireless sensor network system and showed that the hierarchical behavior of the diagram could be correctly represented.

Keywords: CSP, FDR, formal verification, model checking

1 Introduction

UML (Unified Modeling Language) is one of the most popular specification languages and is a de-facto standard modeling language for object-oriented software. While it is less used in software development projects [1], it is used well in model-driven software development and embedded system design [2]. Recently UML is also used to model IoT (Internet of Things) system [3], and especially the use of UML expands in application to modeling wireless sensor networks [4][5][6]. Wireless sensor networks are often documented with UML, and their mechanisms are reviewed on the documents. Such documents comprise

[*] Graduate School of Okayama Prefectural University, Okayama, Japan
[†] Okayama Prefectural University, Okayama, Japan
[‡] Ehime University, Ehime, Japan

multiple models and views related to each other, and their inconsistencies must be managed through system development [7]. The inconsistencies of different models and views of software systems may cause many errors and, therefore may make their management complicated. Developers spend considerable effort to keep the consistency [7][8][9][10], and automatic consistency checking has been expected.

State machine diagram and sequence diagram of UML are widely used to model wireless sensor networks. Sequence diagrams represent interactions between components in a network, and state machine diagrams elaborate behaviors of the components. Here, the state machine diagrams must satisfy the interactions described in the sequence diagrams. Some automatic consistency checking methods for those diagrams have been proposed so far. Zhao et al. [11] proposed a method for checking the consistency between UML sequence and state machine diagrams using SPIN model checker. A scenario described by a sequence diagram is expressed as a "never claim," and SPIN can check whether the state machine diagrams have the behavior as claimed. Egyed [12] provided consistency rules for UML class, sequence, and state machine diagrams. Kaufmann et al. [13] proposed a method to verify whether state machine diagrams can execute a desired or forbidden sce-nario modeled by a sequence diagram using SAT solver. Yokogawa et al. [14] developed a method for inter-model consistency verification of a sequence diagram and state machine diagrams using the FDR model checker [15]. Phuklang et al. [16] developed a tool supporting to conduct consistency verification. Those studies can successfully detect several types of inconsistencies between state machine diagrams and a sequence diagram.

To specify large and complicated wireless sensor networks, first individual functionalities are described as simple models, and then they are integrated into a large model. Such models often have a hierarchical structure. There are several studies that provide formalisms for a state machine diagram with a hierarchical structure. André et al. [17][18] provided a formalization of state machine diagrams with hierarchy (e.g. composite states, internal/external transitions) and concurrent aspects (e.g. orthogonal regions, folk/join transitions) using colored Petri Nets. Zhang et al. [19] provided a translation method of state machine diagrams into CSP#, which is an input language for the PAT model checker [20]. Their translation supports a large subset of state machine diagrams including join/folk transitions, history pseudo-states, entry/exit points, and so on. Concerning sequence diagrams, however, most of the existing formalisms only focus on "flat" sequence diagrams and leave hierarchical (i.e., non-flat) structure out of the scope.

Miyazaki et al. [21] provided a process representation of sequence diagrams with a single combined fragment and a method for refinement checking of sequence diagrams using LTSA [22]. Matsumoto et al. [23] proposed a method for consistency verification of sequence diagrams with multiple combined fragments in a similar manner to [14]. However, these methods support only a few types of combined fragments and cannot represent a complicated control structure of sequence diagrams.

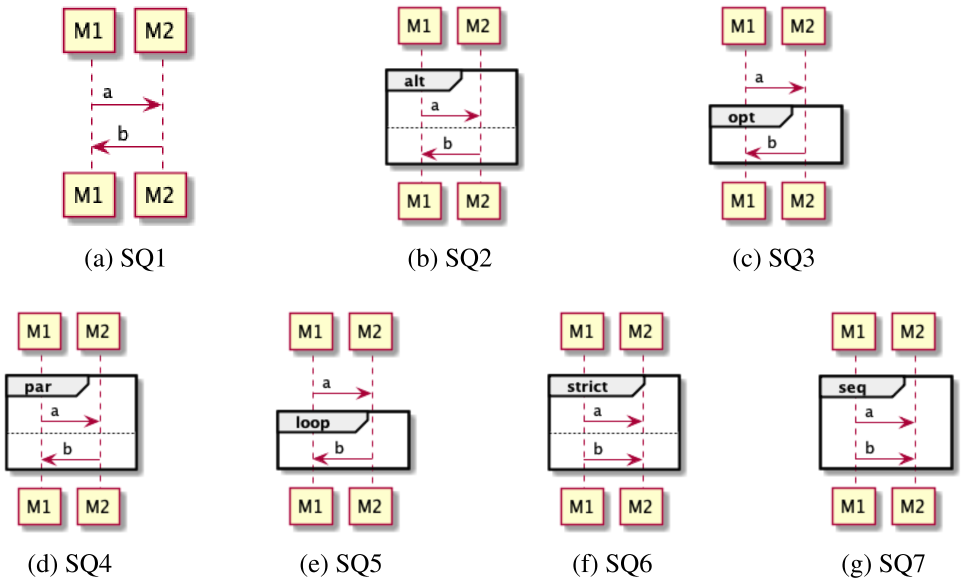
In this paper, we extend the process representation shown in [23] to handle other types of combined fragments describing the hierarchical control structure. Supporting the hierarchical structure makes it possible to guarantee the consistency of software system designs with complicated scenarios. Besides, we simplify the process representation and make it more structured. This modification makes it possible to omit redundant events on the representation of control structure and to describe each interaction fragments as processes individually.

The contributions of this paper are: (1) providing process description of sequence diagrams with a hierarchical structure in CPS_M, (2) supporting six types of combined fragments which can describe complicated control structure of interactions, and (3) providing a case study of process representation for interactions of a wireless sensor network system.

2 Consistency of UML Diagrams

2.1 Sequence Diagrams

A sequence diagram is composed of *lifelines* L, *event occurrences* E, *messages* M, *interaction fragments* F, and *combined fragments* C. A lifeline corresponds to a module of a system. A message represents an interaction between modules. An event occurrence describes sending/receiving of a message or a reference of a combined fragment. Sending and receiving of message m is labeled as m! and m?, and a reference of a combined fragment c is labeled as c*. An interaction fragment is a collection of interactions and is defined as a sequence of event occurrences. A combined fragment is described as a frame over lifelines and represents a control structure of interaction fragments. A combined fragment is composed of an interaction operator and one or more interaction fragments. We now focus on six types of interaction operators defined in UML 2.0, *alt*, *opt*, *par*, *loop*, *strict* and *seq*. *alt* operator represents a non-deterministic selection of interaction fragments. One of the interactions inside the *alt* frame is executed. *opt* operator represents an arbitrary execution of an interaction fragment. It can be decided arbitrarily whether or not to execute the interaction inside the *opt* frame. *par* operator represents a parallel execution of interaction fragments. The interactions of the *par* frame are executed interleavingly. *loop* operator represents an iterative execution of an interaction fragment. The interaction inside the *loop* frame is iteratively executed, and the iteration non-deterministically finishes. *strict* operator represents the strong sequencing of interaction fragments. The interactions inside the *strict* frame are executed in order from top to bottom. *seq* operator represents the weak sequencing of an interaction fragment. The event occurrences on the same lifeline inside the *seq* frame are executed in order from top to bottom.



(a) SQ1 (b) SQ2 (c) SQ3

(d) SQ4 (e) SQ5 (f) SQ6 (g) SQ7

Figure 1: Examples of sequence diagrams

Fig. 1 shows examples of sequence diagrams. Fig. 1 (a) describes a (flat) sequence diagram. Fig. 1 (b) $\sim$ (g) describe sequence diagrams with a combined fragment. $SQi = \langle L^i, E^i, M^i, F^i, C^i \rangle$ ($i = 1 \sim 7$) are defined as follows:

$$
\begin{aligned}
&L^1 = L^2 = L^3 = L^4 = L^5 = L^6 = L^7 = \{M1, M2\},\\
&M^1 = M^2 = M^3 = M^4 = M_5 = M^6 = M_7 = \{a, b\},\\
&E^1 = \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}\}, E^2 = \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}, \mathsf{c1*}\}, E^3 = \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}, \mathsf{c2*}\},\\
&E^4 = \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}, \mathsf{c3*}\}, E^5 = \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}, \mathsf{c4*}\},\\
&E^6 = \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}, \mathsf{c5*}\}, E^7 = \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}, \mathsf{c6*}\},\\
&F^1 = \{f_1\}, F^2 = \{f_2, f_3, f_4\}, F^3 = \{f_5, f_6\}, F^4 = \{f_7, f_8, f_9\}, F^5 = \{f_{10}, f_{11}\},\\
&F^6 = \{f_{12}, f_{13}, f_{14}\}, F^7 = \{f_{15}, f_{16}\} \text{ where}\\
&\quad f_1 = \{\langle M1, (\mathsf{a!b?})\rangle, \langle M2, (\mathsf{a?b!})\rangle\},\\
&\quad f_2 = \{\langle M1, (\mathsf{c1*})\rangle, \langle M2, (\mathsf{c1*})\rangle\}, f_3 = \{\langle M1, (\mathsf{a!})\rangle, \langle M2, (\mathsf{a?})\rangle\},\\
&\quad f_4 = \{\langle M1, (\mathsf{b?})\rangle, \langle M2, (\mathsf{b!})\rangle\},\\
&\quad f_5 = \{\langle M1, (\mathsf{a!c2*})\rangle, \langle M2, (\mathsf{a?c2*})\rangle\}, f_6 = \{\langle M1, (\mathsf{b?})\rangle, \langle M2, (\mathsf{b!})\rangle\},\\
&\quad f_7 = \{\langle M1, (\mathsf{c3*})\rangle, \langle M2, (\mathsf{c3*})\rangle\}, f_8 = \{\langle M1, (\mathsf{a!})\rangle, \langle M2, (\mathsf{a?})\rangle\},\\
&\quad f_9 = \{\langle M1, (\mathsf{b?})\rangle, \langle M2, (\mathsf{b!})\rangle\},\\
&\quad f_{10} = \{\langle M1, (\mathsf{a!c4*})\rangle, \langle M2, (\mathsf{a?c4*})\rangle\}, f_{11} = \{\langle M1, (\mathsf{b?})\rangle, \langle M2, (\mathsf{b!})\rangle\},\\
&\quad f_{12} = \{\langle M1, (\mathsf{c5*})\rangle, \langle M2, (\mathsf{c5*})\rangle\}, f_{13} = \{\langle M1, (\mathsf{a!})\rangle, \langle M2, (\mathsf{a?})\rangle\},\\
&\quad f_{14} = \{\langle M1, (\mathsf{b!})\rangle, \langle M2, (\mathsf{b?})\rangle\},\\
&\quad f_{15} = \{\langle M1, (\mathsf{a!c6*})\rangle, \langle M2, (\mathsf{a?c6*})\rangle\}, f_{16} = \{\langle M1, (\mathsf{b!})\rangle, \langle M2, (\mathsf{b?})\rangle\},\\
&C^1 = \{\}, C^2 = \{c_1\}, C^3 = \{c_2\}, C^4 = \{c_3\}, C^5 = \{c_4\}, C^6 = \{c_5\}, C^7 = \{c_6\} \text{ where}\\
&\quad c_1 = \langle alt, \{f_3, f_4\}\rangle, c_2 = \langle opt, \{f_6\}\rangle, c_3 = \langle par, \{f_8, f_9\}\rangle, c_4 = \langle loop, \{f_{11}\}\rangle,\\
&\quad c_5 = \langle strict, \{f_{13}, f_{14}\}\rangle, c_6 = \langle seq, \{f_{16}\}\rangle.
\end{aligned}
$$

The semantics of sequence diagrams is defined with a set of *computations*, which is a sequence of labels. The sequence diagram in Fig. 1 describe interactions between M1 and M2. The simple sequence diagram SQ1 has one computation a!a?b!b?. Since the *alt* frame describes non-deterministic choice of interactions, SQ2 has two computations a!a? and b!b?. Similarly, SQ3 with the *opt* frame has two computations a!a?b!b? and a!a?. Since the *par* frame describes an interleaving of interactions, SQ4 has the following six computations a!a?b!b?, a!b!a?b?, a!b!b?a?, b!b?a!a?, b!a!b?a?, and b!a!a?b?. Since SQ5 has a *loop* frame, SQ5 has the following infinite computations a!a?, a!a?b!b?, a!a?b!b?b!b?, $\cdots$. Since SQ6 has a *strict* frame, SQ6 has one computation a!a?b!b?. On the other hand, since SQ7 has a *seq* frame, SQ7 has two computations a!a?b!b? and a!b!a?b?.

2.2 Framework of Consistency Verification

The consistency of diagrams is defined as an inclusive relation between the sets of computations [14]. FDR can check a trace inclusion of processes by checking traces refinement relation. By representing sequence diagrams as processes whose traces correspond to their computations, a consistency verification can be done by checking traces refinement of the processes using FDR.

We adopt a similar verification framework to [16] for a sequence diagram and state machine diagrams. Fig. 2 shows the framework for consistency verification using FDR.

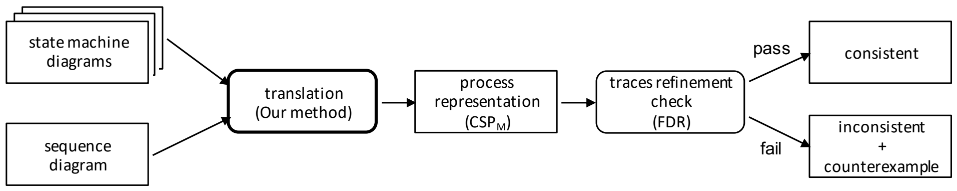

state machine diagrams
sequence diagram
translation (Our method)
process representation (CSP$_M$)
traces refinement check (FDR)
pass
fail
consistent
inconsistent + counterexample

Figure 2: A framework of consistency verification.

These diagrams are translated into a process representation described by CSP$_M$. A model checker FDR takes in the representation and checks a trace refinement relation between the processes of the diagrams. Passing the trace refinement check means that the computations of the sequence diagram are included by those of the state machine diagrams. Therefore, it can be said that the consistency of the diagrams is confirmed. In contrast, failing the check means that the diagrams are inconsistent. In the case of failure, some computations of the sequence diagram are not included by those of the state machine diagrams, and FDR generates a counterexample that shows the computation which is not included. The counterexample can help to fix the inconsistency of diagrams. As stated above, this paper only focuses on the process representation of sequence diagrams.

2.3 CSP$_M$

CSP$_M$ is a functional language for defining CSP (Communicating Sequential Processes) [24] processes, which is supported by FDR tool. We briefly introduce CSP$_M$ notations which are used in the proposed method. Fig. 3 describes an example of CSP$_M$ program.

```
channel a, b
A = {a}

P = a -> Q
Q = b -> SKIP
R = Q; (a -> SKIP)
S = P [] R
T = P [|A|] R
U = Q ||| (a -> SKIP)
```

Figure 3: An example of CSP$_M$ program.

A *channel* is used to create an event and is declared with `channel`. This example has two channels `a` and `b`. Processes are defined by events. "`->`" denotes *prefix* operator and the process `P` performs the event `a` and then `Q` is run. "`SKIP`" denotes a predefined process which immediately terminates. The process `Q` performs the event `b` and then terminates. A *sequential composition* of processes is defined by "`;`" operator. The process `X; Y` behaves like `X` until it terminates and then `Y` is run. In this example, `R` performs the events `b` and `a`, and then terminates. An *external choice* of processes is defined by "`[]`" operator. The process `X[]Y` behaves like either `X` or `Y` according to the initial event of them. In this example, `S` performs either "`a` and `b`" or "`b` and `a`", and then terminates. A *parallel composition* of processes is defined by "`[||]`" operator. The process `X[|A|]Y` runs `X` and `Y` in parallel forcing them to synchronize on the events in `A`. Since `P` and `R` are synchronized by the event a, T first performs b of R, and then performs the synchronized event a, then finally performs b of P and terminates. An *interleave* composition of processes is defined by "|||" operator. The process X|||Y runs X and Y in parallel without any synchronization. In this example, U performs the events a and b in any order. An event can be hidden by *hide* operator "\\." The process X\A behaves like X but when X\A performs the event in A, it is observed as a τ event from other processes.

3 Process Representation

3.1 Overview

A sequence diagram is represented as a process composing processes for its interaction fragments, combined fragments, and messages. Interaction fragments and lifelines are represented as processes that engage in the event occurrences on them. The order of events represents the order relations between event occurrences.

Our method obtains the process representation of a sequence diagram by the following steps:

Step 1. defining sending/receiving of message as a channel (Sec. 3.2),
Step 2. defining begin/end of executing combined fragment as a channel (Sec. 3.2),
Step 3. representing behavior of messages as processes (Sec. 3.3),
Step 4. representing interaction fragments as processes (Sec. 3.4),
Step 5. representing combined fragments as processes (Sec. 3.5), and
Step 6. representing whole sequence diagram as a process (Sec. 3.6).

3.2 Defining Channels

In our method, event occurrences of the sequence diagram are represented as channels. Sending and receiving of message labeled by m! and m? is represented channel ms and mr. Reference of combined fragment labeled by c* is represented two channel cb and ce. The channel cb and ce represents beginning and ending of the combined fragment *c*, respectively. For example, channel definition of SQ2 which has two messages *a*, *b* and one combined fragment c_1 can be obtained as follows:

```
channel as, ar, bs, br
channel c1b, c1e
```

3.3 Representing Messages

For each message *m*, the sending and receiving of *m* labeled by m! and m? have an ordering relation, that is, m! must be executed before m?. Such relation can be represented as a process M = ms->mr->M. By synchronizing the process M, we can remove the paths of the process which violate the order relation over ms and mr. For example, the messages of SQ1 (similar to the other diagrams in Fig. 1) can be represented as the following process MSG:

```
A = as->ar->A
B = bs->br->B
MSG = A ||| B
```

The processes A and B represent messages *a* and *b*. MSG can be obtained as an interleaving composition of the processes for the messages.

3.4 Representing Interaction Fragments

To represent an interaction fragment, we first represent the order relation of event occurrences on each lifeline. And then, we can obtain the process representing the whole interaction fragment by composing the processes for lifelines. We now consider the following interaction fragment f:

$$f \;=\; \{\langle l_1, (\mathsf{e}_{1,1}\mathsf{e}_{1,2}\cdots)\rangle, \langle l_2, (\mathsf{e}_{2,1}\mathsf{e}_{2,2}\cdots)\rangle, \cdots\}$$

where l_i indicates a lifeline and $\mathsf{e}_{i,j}$ indicates the j-th event occurrence on the lifeline l_i. The process representing lifeline l_i can be obtained as the following process $\mathtt{F\_L}i$:

$$\mathtt{F\_L}i \;=\; Prefix(\mathsf{e}_{i,1})\texttt{->}Prefix(\mathsf{e}_{i,2})\texttt{->}\cdots\texttt{->SKIP}.$$

When $\mathsf{e}_{i,j}$ is sending or receiving of message m labeled by m! or m?, $Prefix(\mathsf{e}_{i,1})$ denotes an event `ms` or `mr`, respectively. When $\mathsf{e}_{i,j}$ is a reference of combined fragment c, $Prefix(\mathsf{e}_{i,1})$ denotes a prefix `cb->ce`. The process representation of the interaction fragment f can be obtained as an interleaving of $\mathtt{F\_L}i$ as follows:

$$\mathtt{FI} \;=\; \mathtt{F\_L1}\ \mathtt{|||}\ \mathtt{F\_L2}\ \mathtt{|||}\cdots$$

In the case that f includes some combined fragments c_i on lifeline l_j and l_k, `F_Lj` and `F_Lk`, need to be synchronized by the events `cib` and `cie` as follows:

$$\mathtt{FI} \;=\; \mathtt{F\_L1}\ \mathtt{|||}\cdots\mathtt{|||}\ \mathtt{F\_Lj}\ \mathtt{|[\{cib,\ cie\}]|}\ \mathtt{F\_Lk}\ \mathtt{|||}\cdots$$

This is because the execution of c_i must start simultaneously on the lifelines l_j and l_k.

For example, the process `F1I` for the interaction fragment f_1 of the sequence diagram SQ1 is obtained as follows:

```
F1_M1 = as->br->SKIP
F1_M2 = ar->bs->SKIP
F1I   = F1_M1|||F1_M2
```

By synchronizing `F1I` with `MSG` described above, we can obtain the process representation for SQ1 as follows:

```
SQ1 = F1I [|{as, ar, bs, br}|] MSG
```

3.5 Representing Combined Fragments

3.5.1 alt *operator*

As stated above, a combined fragment is composed of several interaction fragments. The process for a combined fragment can be obtained by composing the processes for interaction fragments according to the interaction operator.

We consider the following combined fragment c with *alt* operator:

$$c = \langle alt, \{f_1, f_2, \cdots, f_n\}\rangle$$

When the execution of combined fragment c starts, one of the interaction fragments $f_1, f_2, \cdots,$ f_n is selected non-deterministically and executed. The execution of the combined fragment c will finish when the execution of the interaction fragment completes. Such behavior can be represented by an external choice operator (`[]`) and a sequential composition operator (`;`). The process representing c is obtained by composing the processes for interaction fragments as follows:

```
CI  =  (cb->(F1I [] F2I []···[] FnI));(ce->CI)
```

By synchronizing the process for c with the process for interaction fragment f referring c, we can represent whole interactions of f as a process.

Consider SQ2, the interaction fragment f_2 includes the combined fragment c_1, and c_1 is composed of f_3 and f_4. The interaction fragment f_2 is represented as the following process `F2I`:

```
F2_M1 = c1b->c1e->SKIP
F2_M2 = c1b->c1e->SKIP
F2I   = F2_M1 [|{c1b, c1e}|] F2_M2
```

The interaction fragments f_3 and f_4 are represented as the following processes `F3I` and `F4I`:

```
F3_M1 = as->SKIP
F3_M2 = ar->SKIP
F3I   = F3_M1 ||| F3_M2
F4_M1 = br->SKIP
F4_M2 = bs->SKIP
F4I   = F4_M1 ||| F4_M2
```

The combined fragment c_1 is represented as the following process `C1I`:

```
C1I = (c1b->(F3I [] F4I));(c1e->C1I)
```

The whole interactions of fragment f_2 including c_1 can be obtained as the following process `F2C1I`:

```
F2C1I = F2I [|{c1b, c1e}|] C1I
```

Finally By synchronizing `F2C1I` with `MSG` described above, we can obtain the process representation for SQ2 as follows:

```
SQ2 = F2C1I [|{as, ar, bs, br}|] MSG\{c1b, c1e}
```

Note that the events `c1b` and `c1e` are hidden. This is because we only focus on the inclusive relation over sending/receiving of messages of diagrams to verify their consistency.

3.5.2 opt *operator*

We consider the following combined fragment c with *opt* operator:

$$c = \langle opt, \{f\}\rangle$$

When the execution of combined fragment c starts, it is decided non-deterministically whether or not the interaction fragments f is executed. Similar to *alt*, such behavior can be represented by an external choice and a sequential composition. The process representing c is obtained by composing the processes for interaction fragments as follows:

```
CI  =  (cb->(FI [] SKIP));(ce->CI)
```

By synchronizing the process for c with the process for interaction fragment f referring c, we can represent whole interactions of f as a process.

Consider SQ3, the interaction fragment f_5 includes the combined fragment c_2, and c_1 is composed of f_6. The interaction fragment f_5 is represented as the following process F5I:

```
F5_M1 = as->c2b->c2e->SKIP
F5_M2 = ar->c2b->c2e->SKIP
F5I   = F5_M1 [|{c2b, c2e}|] F5_M2
```

The interaction fragments f_6 is represented as the following process F6I:

```
F6_M1 = br->SKIP
F6_M2 = bs->SKIP
F6I   = F6_M1 ||| F6_M2
```

The combined fragment c_2 is represented as the following process C2I:

```
C2I = (c2b->(F6I [] SKIP));(c2e->SKIP)
```

The whole interactions of fragment f_5 including c_2 can be obtained as the following process F5C2I:

```
F5C2I = F5I [|{c2b, c2e}|] C2I
```

Finally By synchronizing F5C2I with MSG described above, we can obtain the process representation for SQ4 as follows:

```
SQ4 = F5C2I [|{as, ar, bs, br}|] MSG\{c2b, c2e}
```

3.5.3 par *operator*

We consider the following combined fragment c with *par* operator:

$$c \;=\; \langle par, \{f_1, f_2, \cdots, f_n\}\rangle$$

When the execution of combined fragment c starts, all interaction fragments $f_1, f_2, \cdots, f_n$ are interleavingly executed. The execution of the combined fragment c will finish when the execution of the interaction fragment completes. Such behavior can be represented by an interleave operator (|||). The process representing c is obtained by composing the processes for interaction fragments as follows:

```
CI = (cb->(F1I ||| F2I |||···||| FnI));(ce->CI)
```

By synchronizing the process for c with the process for interaction fragment f referring c, we can represent whole interactions of f as a process.

Consider SQ4, the interaction fragment f_7 includes the combined fragment c_3, and c_3 is composed of f_8 and f_9. The interaction fragment f_7 is represented as the following process F7I:

```
F7_M1 = c3b->c3e->SKIP
F7_M2 = c3b->c3e->SKIP
F7I   = F7_M1 [|{c3b, c3e}|] F7_M2
```

The interaction fragments f_8 and f_9 are represented as the following processes F8I and F9I:

```
F8_M1 = as->SKIP
F8_M2 = ar->SKIP
F8I   = F8_M1 ||| F8_M2
F9_M1 = br->SKIP
F9_M2 = bs->SKIP
F9I   = F9_M1 ||| F9_M2
```

The combined fragment c_3 is represented as the following process `C3I`:

```
C3I = (c3b->(F8I ||| F9I));(c3e->C3I)
```

The whole interactions of fragment f_7 including c_3 can be obtained as the following process `F7C3I`:

```
F7C3I = F7I [|{c3b, c3e}|] C3I
```

Finally By synchronizing `F7C3I` with `MSG` described above, we can obtain the process representation for SQ3 as follows:

```
SQ3 = F7C3I [|{as, ar, bs, br}|] MSG \ {c3b, c3e}
```

3.5.4 loop *operator*

We consider the following combined fragment c with *loop* operator:

$$c = \langle loop, \{f\}\rangle$$

When the execution of combined fragment c starts, the interactions of f is arbitrarily repeated. Such behavior can be represented by an external choice and a sequential composition. The process representing c is obtained by composing the processes for interaction fragments as follows:

```
CI = cb->CL
CL = ((FI;CL)[](ce->CI)
```

By synchronizing the process for c with the process for interaction fragment f referring c, we can represent whole interactions of f as a process.

Consider SQ5, the interaction fragment f_{10} includes the combined fragment c_4, and c_4 is composed of f_{11}. The interaction fragment f_{10} is represented as the following process `F10I`:

```
F10_M1 = as->c4b->c4e->SKIP
F10_M2 = ar->c4b->c4e->SKIP
F10I   = F10_M1 [|{c4b, c4e}|] F10_M2
```

The interaction fragments f_{11} is represented as the following process `F11I`:

```
F11_M1 = br->SKIP
F11_M2 = bs->SKIP
F11I   = F11_M1 ||| F11_M2
```

The combined fragment c_4 is represented as the following process `C4I`:

```
C4I = c4b->C4L
C4L = (F11I;C4L)[](c4e->C4I)
```

The whole interactions of fragment f_{10} including c_4 can be obtained as the following process `F10C4I`:

```
F10C4I = F10I [|{c4b, c4e}|] C4I
```

Finally By synchronizing `F10C4I` with `MSG` described above, we can obtain the process representation for SQ5 as follows:

```
SQ5 = F10C4I [|{as, ar, bs, br}|] MSG \ {c4b, c4e}
```

3.5.5 strict *operator*

We consider the following combined fragment c with *strict* operator:

$$c \;=\; \langle strict, \{f_1, f_2, \cdots, f_n\}\rangle$$

When the execution of combined fragment c starts, all interaction fragments $f_1, f_2, \cdots, f_n$ are executed in order from f_1 to f_n. The execution of the combined fragment c will finish when the execution of the interaction fragment completes. Such behavior can be represented by a sequential composition operator (;). The process representing c is obtained by composing the processes for interaction fragments as follows:

```
CI  =  (cb->F1I); F2I;···; FnI;(ce->CI)
```

By synchronizing the process for c with the process for interaction fragment f referring c, we can represent whole interactions of f as a process.

Consider SQ6, the interaction fragment f_{12} includes the combined fragment c_5, and c_5 is composed of f_{13} and f_{14}. The interaction fragment f_{12} is represented as the following process F12I:

```
F12_M1 = c5b->c5e->SKIP
F12_M2 = c5b->c5e->SKIP
F12I   = F12_M1 [|{c5b, c5e}|] F12_M2
```

The interaction fragments f_{13} and f_{14} are represented as the following processes F13I and F14I:

```
F13_M1 = as->SKIP
F13_M2 = ar->SKIP
F13I   = F13_M1 ||| F13_M2
F14_M1 = bs->SKIP
F14_M2 = br->SKIP
F14I   = F14_M1 ||| F14_M2
```

The combined fragment c_5 is represented as the following process C5I:

```
C5I = c5b->F13I; F14I; c5e->C5I
```

The whole interactions of fragment f_12 including c_5 can be obtained as the following process F12C5I:

```
F12C5I = F12I [|{c5b, c5e}|] C5I
```

Finally By synchronizing F12C5I with MSG described above, we can obtain the process representation for SQ5 as follows:

```
SQ5 = F12C5I [|{as, ar, bs, br}|] MSG \ {c5b, c5e}
```

3.5.6 seq *operator*

We consider the following combined fragment c with *seq* operator:

$$c \;=\; \langle seq, \{f\}\rangle$$

Since the behavior of combined fragment c is similar to the behavior of the interaction fragments f inside c, the process representing c is obtained by composing the processes for interaction fragments as follows:

```
CI  =  (cb->FI);(ce->CI)
```

By synchronizing the process for c with the process for interaction fragment f referring c, we can represent whole interactions of f as a process.

Consider SQ7, the interaction fragment f_{15} includes the combined fragment c_6, and c_6 is composed of f_{16}. The interaction fragment f_{15} is represented as the following process `F15I`:

```
F15_M1  = c6b->c6e->SKIP
F15_M2  = c6b->c6e->SKIP
F15I    = F15_M1 [|{c6b, c6e}|] F15_M2
```

The interaction fragments f_{16} is represented as the following process `F16I`:

```
F16_M1  = as->bs->SKIP
F16_M2  = ar->br->SKIP
F16I    = F16_M1 ||| F16_M2
```

The combined fragment c_6 is represented as the following process `C6I`:

```
C6I = c6b->F16I; c6e->C6I
```

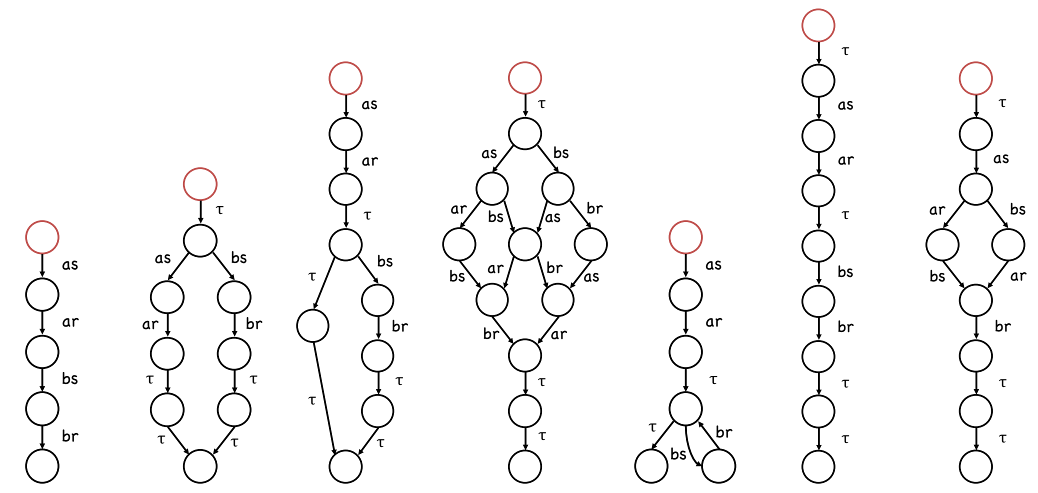
The whole interactions of fragment f_{15} including c_6 can be obtained as the following process `F15C6I`:

```
F15C6I = F15I [|{c6b, c6e}|] C6I
```

Finally By synchronizing `F15C6I` with `MSG` described above, we can obtain the process representation for SQ6 as follows:

```
SQ6 = F15C6I [|{as, ar, bs, br}|] MSG \ {c6b, c6e}
```

Fig. 4 shows the graphs of the processes `SQ1`, `SQ2`, `SQ3`, `SQ4`, `SQ5`, `SQ6` and `SQ7` generated by FDR tool.



(a) `SQ1` (b) `SQ2` (c) `SQ3` (d) `SQ4` (e) `SQ5` (f) `SQ6` (g) `SQ7`

Figure 4: The graphs of the processes generated by FDR.

3.6 Representing Interactions of Whole Sequence Diagram

As stated in Sec. 3.5, we can obtain the process representing whole interactions of f including c by synchronizing the processes for f and c. When a sequence diagram has nested structure, processes for combined fragments are composed step-by-step from the deepest one. Processes that have nest relation are synchronized by the begin/end events of the deeper one.

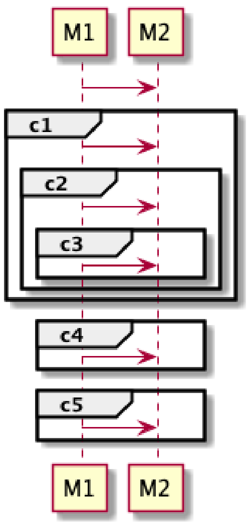


Figure 5: A sequence diagram with a hierarchical structure.

Consider the sequence diagram shown in Fig. 5 which has the topmost interaction fragment (referring f) and combined fragments c_1, c_2, c_3, c_4, c_5. Since c_1, c_2 and c_3 have nested structure and the deepest fragment is c_3, we first synchronize the processes C2I and C3I by the begin/end events of c_3 as follows:

```
C2C3I = C2I[|{c3b, c3e}|]C3I
```

Then the process C1I and the obtained process C2C3I are synchronized by the begin/end events of c_2 as follows:

```
C1C2C3I = C1I[|{c2b, c2e}|]C2C3I
```

Processes in the same depth are synchronized without begin/end events. The processes C4I, C5I and C1C2C3I are synchronized as follows:

```
C4C5I = C4I ||| C5I
C1C2C3C4C5I = C1C2C3I ||| C4C5I
```

Finally, the process FI and C1C2C3C4C5I are synchronized as follows:

```
FC1C2C3C4C5
  = FI[|{c1b, c1e, c4b, c4e, c5b, c5e}|]C1C2C3C4C5I
```

4 A Case Study

We applied the proposed method for process representation to a sequence diagram shown in [5]. Fig. 6 shows the sequence diagram, which describes interactions of basic components of a wireless sensor network system. This sequence diagram has two *loop* combined fragments as a nested structure. This system has six components: Mediator, Power Scaling Monitor (Monitor), Power Scaling Controller (Reasoner), Power Transmission Actuator

(Actuator), Node Degree Observation (NDObserver) and Battery Level Observation (BattObserver). This sequence diagram depicts the process of monitoring neighbor sensor nodes and conducting appropriate power transmission.

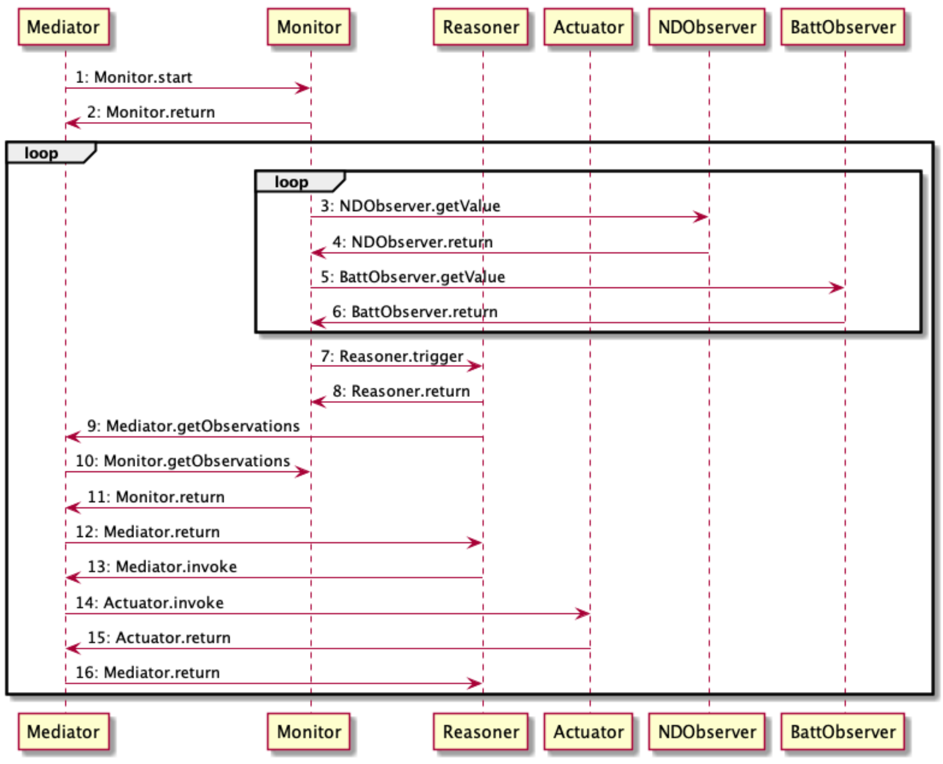


Figure 6: A sequence diagram describing interactions of a wireless sensor network [5].

The Mediator component first starts the Monitor component. The Monitor periodically measures the number of neighbor nodes and the battery level of the nodes. According to the measured values, the component decides whether or not to trigger the Reasoner component. The Reasoner will request the latest measured values to the Mediator. The Mediator captures the request and gets the data from the Monitor and finally returns the data to the Reasoner. Then, the Reasoner will send the request to modify the transmission power of nodes to the Mediator. The Mediator captures the request and invokes the Actuator, and the Actuator handles the request.

According to the proposed process representation, the sequence diagram is encoded to CSP$_M$ description shown in Fig. 7. FDR generates the graph for the process `SQ` shown in Fig. 8. As shown in Fig. 8, the nested iterating behavior described by *loop* is correctly represented in the graph.

5 Summary and Future Remarks

In this paper, we provided a method for the process representation of sequence diagrams. The proposed process representation is suited for consistency verification with state machine diagrams, which is proposed in [14]. Our method supports six types of combined fragments which are used to describe the complicated control structure of interaction fragments. It also can handle sequence diagrams with a hierarchical structure. We showed a case study on which we translate a sequence diagram depicting interactions in a wireless sensor network into CSP$_M$ description. The CSP$_M$ description can be analyzed by FDR tool, and we can successfully obtain the process representation of the sequence diagram.

In this paper, we provided the process description for six types of combined fragments describing a structure of interactions. We will extend our method to handle the other combined fragment notations of sequence diagrams as future work. The framework of consistency verification we proposed can detect inconsistency of diagrams using FDR. In the case that inconsistency is detected, FDR provides a counterexample that shows evidence for violation of trace inclusion relation. Supporting inconsistency fixing using a counterexample generated by FDR is our future plan.

Acknowledgments

This research was partially supported by JST CREST Grant Number JPMJCR1531, Japan.

References

[1] M. Petre, "UML in practice," *Proc. on Int'l Conf. on Softw. Eng. (ICSE 2013)*, pp. 722–731, 2013.

[2] D. Torre, Y. Labiche, and M. Genero, "UML Consistency Rules: A Systematic Mapping Study," *Proc. of the 18th Int'l Conf. on Evaluation and Assessment in Softw. Eng., (EASE 2014)*, pp. 1–10, 2014.

[3] K. Thramboulidis and F. Christoulakis, "UML4IoT - UML-based approach to exploit IoT in cyber-physical manufacturing systems," *Computers in Industry*, vol. 82, pp. 259–272, 2016.

[4] B. Wang and J. S. Baras, "Integrated modeling and simulation framework for wireless sensor networks," in *2012 IEEE 21st Int'l Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 2012, pp. 268–273.

[5] V. H. Díaz, J.-F. Martínez, N. L. Martínez, and R. M. Del Toro, "Self-Adaptive Strategy Based on Fuzzy Control Systems for Improving Performance in Wireless Sensors Networks," *Sensors*, vol. 15, no. 9, pp. 24 125–24 142, sep 2015. [Online]. Available: http://www.mdpi.com/1424-8220/15/9/24125

[6] S. Uke and R. Thool, "UML Based Modeling for Data Aggregation in Secured Wireless Sensor Network," *Procedia Comp. Sci.*, vol. 78, pp. 706–713, 2016, 1st Int'l Conf. on Inf. Security and Privacy 2015.

[7] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille, "Consistency Problems in UML-Based Software Development," in *UML Modeling Languages and Applications, UML 2004 Satellite Activities*, 2005, pp. 1–12.

[8] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009.

[9] F. ul Muram, H. Tran, and U. Zdun, "Systematic Review of Software Behavioral Model Consistency Checking," *ACM Computing Surveys*, vol. 50, no. 2, pp. 1–39, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3071073.3037755

[10] D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A systematic identification of consistency rules for UML diagrams," *Journal of Systems and Software*, vol. 144, no. June, pp. 121–142, 2018.

[11] X. Zhao, Q. Long, and Z. Qiu, "Model Checking Dynamic UML Consistency," in *Proc. of the 8th Int'l Conf. on Formal Engineering Methods, ICFEM 2006*, 2006, Book, pp. 440–459.

[12] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Trans. Softw. Eng.*, vol. 37, pp. 188–204, March 2011.

[13] P. Kaufmann, M. Kronegger, A. Pfandler, M. Seidl, and M. Widl, "A SAT-Based Debugging Tool for State Machines and Sequence Diagrams," in *Proc. of the7th Int'l Conf. on Softw. Language Eng. (SLE 2014)*, 2014, pp. 21–40.

[14] T. Yokogawa, S. Amasaki, K. Okazaki, Y. Sato, K. Arimoto, and H. Miyazaki, "Consistency verification of UML diagrams based on process bisimulation (fast abstract)," in *Proc. of the 19th IEEE Pacific Rim Int'l Symp. on Dependable Computing (PRDC'13)*, 2013, pp. 126–127.

[15] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "FDR3: a parallel refinement checker for CSP," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 2, pp. 149–167, 2016.

[16] S. Phuklang, T. Yokogawa, P. Leelaprute, and K. Arimoto, "Tool Support for Consistency Verification of UML Diagrams," in *Poster Session on The 18th Int'l Conf. on Product-Focused Softw. Process Improvement (Profes 2017)*, 2017.

[17] É. André, C. Choppy, and K. Klai, "Formalizing non-concurrent UML state machines using colored petri nets," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 4, pp. 1–8, 2012.

[18] É. André, M. M. Benmoussa, and C. Choppy, "Formalising concurrent UML state machines using coloured Petri nets," *Formal Aspects of Computing*, vol. 28, no. 5, pp. 805–845, 2016.

[19] S. J. Zhang and Y. Liu, "An automatic approach to model checking UML state machines," in *Proc. of the 4th Int'l Conf. on Secure Softw. Integration and Reliability Improvement Companion (SSIRI-C)*, 2010, pp. 1–6.

[20] J. Sun, Y. Liu, and J. S. Dong, "Model checking CSP revisited: Introducing a process analysis toolkit," in *Proc. International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, vol. 17 CCIS, 2008, pp. 307–322.

[21] H. Miyazaki, T. Yokogawa, S. Amasaki, K. Asada, and Y. Sato, "Synthesis and refinement check of sequence diagrams." *IEICE Trans. on Inf. and Syst.*, vol. E95-D, no. 9, pp. 2193–2201, 2012.

[22] J. Magee and J. Kramer, *Concurrency: State Models & Java Programs*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[23] A. Matsumoto, T. Yokogawa, S. Amasaki, K. Arimoto, and H. Aman, “Consistency Verification of UML Sequence Diagrams Modeling Wireless Sensor Networks,” in *Proc. 8th International Congress on Advanced Applied Informatics (IIAI-AAI 2019)*, 2019, pp. 458–461.

[24] C. A. R. Hoare, “Communicating sequential processes,” *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

```
channel sm1, rm1, sm2, rm2, sm3, rm3, sm4, rm4, sm5, rm5,
  sm6, rm6, sm7, rm7, sm8, rm8, sm9, rm9, sm10, rm10, sm11, rm11,
  sm12, rm12, sm13, rm13, sm14, rm14, sm15, rm15, sm16, rm16
channel c1b, c1e, c2b, c2e
msg = {sm1, rm1, sm2, rm2, sm3, rm3, sm4, rm4, sm5, rm5, sm6, rm6,
  sm7, rm7, sm8, rm8, sm9, rm9, sm10, rm10, sm11, rm11, sm12, rm12,
  sm13, rm13, sm14, rm14, sm15, rm15, sm16, rm16}

F1_Mediator    = sm1->rm2->c1b->c1e->SKIP
F1_Monitor     = rm1->sm2->c1b->c1e->SKIP
F1_Reasoner    = c1b->c1e->SKIP
F1_Actuator    = c1b->c1e->SKIP
F1_NDObserver = c1b->c1e->SKIP
F1_BattObserver = c1b->c1e->SKIP
F1I = F1_Mediator [|{c1b, c1e}|] (F1_Monitor [|{c1b, c1e}|]
  (F1_Reasoner [|{c1b, c1e}|] (F1_Actuator [|{c1b, c1e}|]
  (F1_NDObserver [|{c1b, c1e}|] F1_BattObserver))))

F2_Mediator    = rm9->sm10->rm11->sm12->rm13->sm14->rm15->sm16->SKIP
F2_Monitor     = c2b->c2e->sm7->rm8->rm10->sm11->SKIP
F2_Reasoner    = c2b->c2e->rm7->sm8->sm9->rm12->sm13->rm16->SKIP
F2_Actuator    = c2b->c2e->rm14->sm15->SKIP
F2_NDObserver = c2b->c2e->SKIP
F2_BattObserver = c2b->c2e->SKIP
F2I = F2_Mediator ||| (F2_Monitor [|{c2b, c2e}|]
  (F2_Reasoner [|{c2b, c2e}|] (F2_Actuator [|{c2b, c2e}|]
  (F2_NDObserver [|{c2b, c2e}|] F2_BattObserver))))

F3_Monitor     = sm3->rm4->sm5->rm6->SKIP
F3_NDObserver = rm3->sm4->SKIP
F3_BattObserver = rm5->sm6->SKIP
F3I = F3_Monitor ||| (F3_NDObserver ||| F3_BattObserver)

C1I = c1b->C1L
C1L = ((F2I;C1L)[](c1e->C1I))

C2I = c2b->C2L
C2L = ((F3I;C2L)[](c2e->C2I))

C1C2I = C1I [|{c2b, c2e}|] C2I
F1C1C2I = F1I [|{c1b, c1e}|] C1C2I

M1 = sm1->rm1->M1
M2 = sm2->rm2->M2
M3 = sm3->rm3->M3
M4 = sm4->rm4->M4
M5 = sm5->rm5->M5
M6 = sm6->rm6->M6
M7 = sm7->rm7->M7
M8 = sm8->rm8->M8
M9 = sm9->rm9->M9
M10 = sm10->rm10->M10
M11 = sm11->rm11->M11
M12 = sm12->rm12->M12
M13 = sm13->rm13->M13
M14 = sm14->rm14->M14
M15 = sm15->rm15->M15
M16 = sm16->rm16->M16

MSG = M1 ||| M2 ||| M3 ||| M4 ||| M5 ||| M6 ||| M7 ||| M8 |||
  M9 ||| M10 ||| M11 ||| M12 ||| M13 ||| M14 ||| M15 ||| M16

SQ = F1C1C2I [|msg|] MSG \ {c1b, c1e, c2b, c2e}
```

Figure 7: CSP$_M$ description.

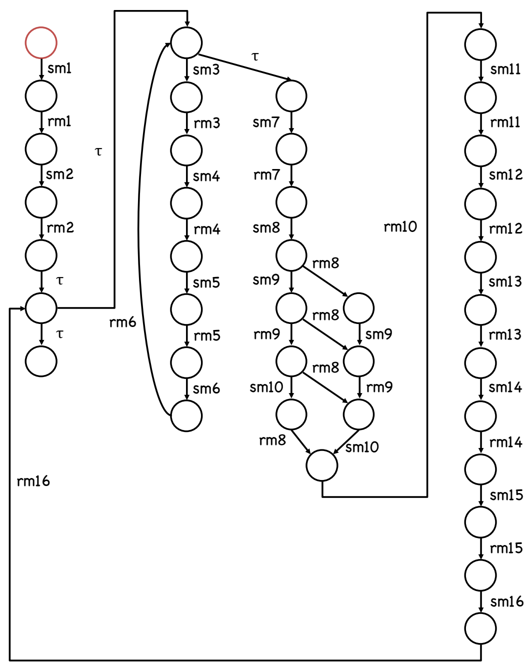


Figure 8: Graph for the process `SQ` by FDR.