ELSEVIER

Information Processing Letters 84 (2002) 125–130

Information Processing Letters

www.elsevier.com/locate/ipl

# A semantics of sequence diagrams [☆]

Seung Mo Cho [*], Hyung Ho Kim, Sung Deok Cha, Doo Hwan Bae

*Department of Computer Science and Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-dong, Yusong-gu, Taejon, South Korea*

Received 10 November 2000; received in revised form 23 February 2002

Communicated by M. Yamashita

**Abstract**

We develop a formal semantics of sequence diagrams. The semantics is given in terms of our new temporal logic, named HDTL, which is designed to specify dynamically evolving systems. This approach allows to facilitate the generic feature of sequence diagrams as well as an automatic analysis, the identification of the instances of a sequence diagram over a trace.
© 2002 Elsevier Science B.V. All rights reserved.

*Keywords:* Formal semantics; Specification languages; Sequence diagram; Temporal logic

## 1. Introduction

Sequence diagrams and Message Sequence Charts (MSCs) offer an intuitive and visual way of describing design requirements by illustrating possible message exchanges among objects in software systems. Recently, integration of sequence diagrams into UML by OMG and standardization of MSC by ITU show that these techniques are widely accepted in practice.

There have been various efforts on the formalization and analysis of MSCs. Examples include approaches based on automata theory [6], Petri-net theory [1,4], and process algebra [7]. Most of them assume that the semantics of sequence diagrams are equivalent to that of MSCs. However, there is a significant difference between sequence diagrams and MSCs: sequence diagrams can be *generic* in the sense that they can describe interactions among arbitrary objects, i.e., participating objects are not fixed.

Let us consider this further. Suppose that there be a sequence diagram $S$ that describes an interaction in a typical client/server system. In general, there are many client objects that may interact with a server object. In this case, the intent of the sequence diagram $S$ is to describe the interactions among *all* possible combinations. Unfortunately, proposed semantic definitions of MSCs do not address the generic case. To cope with this case, an underlying formalism requires the notion of *variables* that capture arbitrary objects from a possible domain and should provide a mechanism for introducing variables, say *quantifier*.

Even though classical quantifiers such as $\forall$ and $\exists$ provide a precise means to introduce variables, we argue that they are inappropriate for automatic analysis: in object-oriented systems, it is impossible

[☆] This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

[*] Corresponding author.
*E-mail address:* seung@salmosa.kaist.ac.kr (S.M. Cho).

0020-0190/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0020-0190(02)00270-3

to fix a set of objects because the configuration of objects changes over time. However, the definitions of classical quantifiers require such sets. This makes it a challenging problem to check the validity of a formula employing quantifiers only over a trace.

The goal of this paper is to develop a semantics of sequence diagrams to facilitate its generic feature as well as automatic analysis. Our semantics is based on a variant of temporal logic, named *Half-order Dynamic Temporal Logic* (HDTL) [2], employing the *freeze quantifier* proposed in [5]. The novelty of freeze quantifier comes from its definition that is only based on a trace. That is, the definition of freeze quantifier allows the evaluation of a temporal formula including variables over a trace, without the notion of a fixed set of objects.

Our semantics allows us to identify the instances of a sequence diagram over a trace using the analysis technique, called *tableau method* [3]. For a given temporal logic formula $f$, tableau method constructs an automaton that accepts only a set of traces over which $f$ holds. Unfortunately, when a temporal formula includes variables, the size of an automaton should be infinite because its alphabet is infinite. This problem motivates the design of intermediate representation, named *flow tree*, that changes its structure on demand. For a given trace, a flow tree identifies a sequence of events that makes the corresponding HDTL formula $f$ hold. When the formula $f$ is the characterization of a sequence diagram $S$, then the identified sequence of events exactly forms $S$'s instance. Note that it requires large amount of effort to distinguish a sequence of causally related events in a trace by hand. Traces usually consist of a large set of unrelated events and, thus, automatic identification is valuable. To the best of our knowledge, this paper is the first attempt to analyze the generic nature of dynamic systems automatically.

## 2. Sequence diagrams

Sequence diagrams illustrate how objects interact with each other. Fig. 1 shows a sample sequence diagram. Vertical lines in the diagram correspond to objects and arrows between vertical lines represent messages exchanged between corresponding objects.

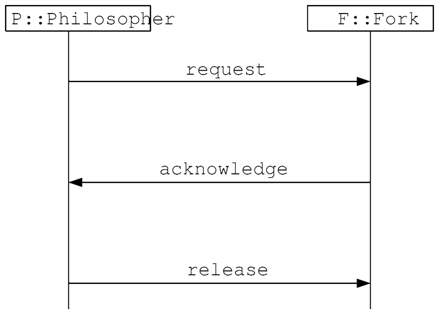

Fig. 1. The sequence diagram of the interaction between a philosopher and a fork.

| No. | Message Label | Sender | Receiver |
|---|---|---|---|
| 1 | request | philosopher #1 | fork #1 |
| 2 | request | philosopher #2 | fork #2 |
| 3 | acknowledge | fork #1 | philosopher #1 |
| 4 | release | philosopher #1 | fork #1 |
| 5 | acknowledge | fork #2 | philosopher #2 |
| 6 | release | philosopher #2 | fork #2 |
| ⋮ | ⋮ | ⋮ | ⋮ |

Fig. 2. A sample trace.

Each arrow is associated with a message label. Time proceeds from top to down.

The simplicity of sequence diagrams make them suitable for expressing required behaviors. However, the interpretation of a sequence diagram over a trace is rather complicated because of the *dynamic nature* of object-oriented systems. Let us consider the following trace (Fig. 2). In this trace, there are two possible *instances* of Fig. 1: one is $\langle 1, 3, 4\rangle$ and the other is $\langle 2, 5, 6\rangle$. It should be noted that the messages 1 and 2 involve in different instances even though they share the same message label, say request. What makes this difference is the participation of objects, called "sender" and "receiver", in each message. Thus, it is impossible to determine which events constitute an instance of a sequence diagram without the information of their participating objects. In addition, the interleaving nature of traces makes it difficult to identify a sequence of related events.

For the sake of simplicity, we focus on relevant features of sequence diagrams. For example, we interpret messages *synchronously*, that is, the delivery of messages and their receipt occur at the same time. Formally, a sequence diagram $S$ is represented by a labeled directed acyclic graph with the following components:

- *Roles:* A finite set $R$ of roles.
- *Messages:* A finite set $M$ of messages.
- *Message labels:* A labeling function $g$ that maps each message in $M$ to a triple $(l, s, r)$ where $l$ denotes a *label* of the message and, $s$ and $r$ denote roles in $R$, called *sender* and *receiver*, respectively.
- *Visual order:* There is partial order $<$ over the messages in $M$. This order $<$ is induced from the local total order $<_r$ over the messages $M$ as following manners:

  $m < m' \quad$ when $m <_r m'$,

  where $m <_r m'$ holds when $m'$ is below $m$ and $s_M(m) = s_M(m')$ or $r_M(m) = r_M(m')$.

For clarity, $l_M(m)$ denotes the label of $g(m)$, and $s_M(m)$ and $r_M(m)$ do its sender and receiver, respectively.

We view the behavior of system under development as a set of finite sequences of events, called traces, where each event denotes the *occurrence* of a message. Assume that there are an infinite set $O$ and a finite set $L$ for participating objects and labels of messages, respectively. Formally, an event $e \in E$ is triple $\langle \mathit{label}, \mathit{sender}, \mathit{receiver}\rangle$ where $\mathit{label} \in L$ is the label of the occurred message, denoted by $l_E(e)$, and $\mathit{sender} \in O$ and $\mathit{receiver} \in O$ are its sender object and receiver object, denoted by $s_E(e)$ and $r_E(e)$, respectively. An event $e$ is said to be the *occurrence* of a message $m$ when $l_E(e) = l_M(m)$. Throughout this paper, $\sigma_0$ denotes the first element of a trace $\sigma$ and $\sigma^i$ denotes the trace that results from $\sigma$ by deleting the first $i$ elements.

## 3. Temporal logic-based semantics

In this section, we introduce HDTL and develop a semantics of sequence diagrams by defining the *semantic function* that translates a sequence diagram $S$ to an HDTL formula $f$.

Assume that there be a set $V$ for variables and a set $L$ for message labels. The formulae of HDTL are built from proposition symbols by Boolean connectives, temporal operators, and freeze quantifiers.

**Definition 1** (*Syntax of HDTL*). Term $\pi$ and formula $\phi$ of HDTL are inductively defined as follows:

- $\pi := s(x) \mid r(x) \mid l(x) \mid \mathsf{l}$
- $\phi := \pi_1 = \pi_2 \mid \mathsf{true} \mid \phi_1 \rightarrow \phi_2 \mid \neg\phi \mid \bigcirc\phi \mid \odot\phi \mid \Diamond\phi \mid x.\phi$

where $x \in V$ and $\mathsf{l} \in L$.

In $x.\phi$, the variable $x$ is bound by the freeze quantifier "$x.$" to the given event in which $x.\phi$ is evaluated. The meanings of the others and the abbreviations (e.g., $a \wedge b \equiv \neg(a \rightarrow \neg b)$) are as usual (cf. [3]). The following definition of semantics captures this idea.

**Definition 2** (*Semantics of HDTL*). Let $\sigma$ be a trace and $\mathcal{E}: V \rightarrow E$ an environment for variables. The HDTL formula $\phi$ is said to be satisfied by a pair $(\sigma, \mathcal{E})$ when $\sigma \models_{\mathcal{E}} \phi$, where the satisfaction relation $\models$ is inductively defined as follows:

- $\sigma \models_{\mathcal{E}} \pi_1 = \pi_2$ *iff* $\mathcal{E}(\pi_1) = \mathcal{E}(\pi_2)$,
- $\sigma \models_{\mathcal{E}} \mathsf{true}$,
- $\sigma \models_{\mathcal{E}} \phi_1 \rightarrow \phi_2$ *iff* $\sigma \models_{\mathcal{E}} \phi_1$ *implies* $\sigma \models_{\mathcal{E}} \phi_2$,
- $\sigma \models_{\mathcal{E}} \neg\phi$ *iff* $\sigma \not\models_{\mathcal{E}} \phi$,
- $\sigma \models_{\mathcal{E}} \bigcirc\phi$ *iff* $|\sigma| > 1 \rightarrow \sigma^1 \models_{\mathcal{E}} \phi$,
- $\sigma \models_{\mathcal{E}} \odot\phi$ *iff* $|\sigma| > 1 \wedge \sigma^1 \models_{\mathcal{E}} \phi$,
- $\sigma \models_{\mathcal{E}} \Diamond\phi$ *iff* $\sigma^i \models_{\mathcal{E}} \phi$ *for* $\exists i, 0 \leqslant i < |\sigma|$,
- $\sigma \models_{\mathcal{E}} x.\phi$ *iff* $\sigma \models_{\mathcal{E}[x:=\sigma_0]} \phi$,

where $\mathcal{E}$ denotes the environment to map a variable $v \in \mathcal{V}$ to an event $e \in E$.

In this definition, $\mathcal{E}(f(x))$ means $f_E(\mathcal{E}(x))$ and $\mathcal{E}(\mathsf{l})$ is just $\mathsf{l}$. $\mathcal{E}[x := \sigma]$ results in a new environment that maps $x$ to $\sigma$ and the others to the same as those of $\mathcal{E}$.

Note that the meaning of a freeze quantifier is defined in terms of a trace only. Thus, when a formula is *closed*, that is, all occurrences of variables are within the scope of corresponding freeze quantifiers, its truth value is completely determined by a trace. Without the

```
fun $sem(R, M, <) =$
  $traces := \{\langle(v_1, m_1), \ldots, (v_m, m_m)\rangle$: for $\forall\langle m_1, m_2, \ldots, m_m\rangle \in seq(M, <)$
                                                    such that $v_i$ are distinct variables for $1 \leqslant i \leqslant m\}$;
  return $\forall t \in traces$: “$\bigvee \Diamond$” $formula(t, [\forall r \in R\colon\ r \mapsto \bot])$;

  where fun $formula(t, env) =$
    if $t = \langle(v, m)\rangle$ then return $v$“.(”$node(t_0, env)$“)”
    else return $v$“.(”$node(t_0, env)$“$\wedge \odot \Diamond$” $formula(t^1, env)$“)”

  and fun $node(\langle v, m\rangle, env) =$
    if $env(s_M(m)) = \bot \wedge env(r_M(m)) = \bot$ then
      $env := env \oplus [s_M(m), r_M(m) \mapsto s(v), r(v)]$;
      return “$l($”$v$“$) =$”$l_M(m)$;
    else if $env(s_M(m)) \neq \bot \wedge env(r_M(m)) = \bot$ then
      $env := env \oplus [r_M(m) \mapsto r(v)]$;
      return “$l($”$v$“$) =$”$l_M(m)$“$\wedge s($”$v$“$) =$”$env(s_M(m))$;
    else if $env(s_M(m)) = \bot \wedge env(r_M(m)) \neq \bot$ then
      $env := env \oplus [s_M(m) \mapsto s(v)]$;
      return “$l($”$v$“$) =$”$l(m)$“$\wedge r($”$v$“$) =$”$env(r_M(m))$;
    else (* $env(s_M(m)) \neq \bot \wedge env(r_M(m)) \neq \bot$ *)
      return “$l($”$v$“$) =$”$l(m)$“$\wedge s($”$v$“$) =$”$env(s_M(m))$“$\wedge r($”$v$“$) =$ ”$env(r_M(m))$;
```

Fig. 3. The definition of semantic function.

loss of generality, we assume that every formula be closed hereinafter.

Fig. 3 shows the definition of semantic function where $\oplus$ means the function overloading and the function $seq(M, <)$ generates a set containing every permutations of messages from $M$ without violating the partial order $<$. In this definition, the function *sem* generates a string as a HDTL formula and the juxtaposition of strings means the string concatenation. For example, we obtain the following HDTL formula by applying the function *sem* to the sequence diagram in Fig. 1.

$$F_1 = \Diamond v_1.\big(l(v_1) = \text{request} \wedge \odot\Diamond v_2.\big(l(v_2) = \text{acknowledge} \wedge s(v_2) = r(v_1) \wedge r(v_2) = s(v_1) \wedge \odot\Diamond v_3.\big(l(v_3) = \text{release} \wedge s(v_3) = s(v_1) \wedge r(v_3) = r(v_1)\big)\big)\big).$$

Let us explain the definition from inner to outer functions: for a message $m$, the function *node* generates a clause capturing constraint on role and message label. That is, for an event $e$ “fixed” by a variable $v$, the resulting clause asserts that the label of $e$ be the same as that of $m$, and the sender and receiver of $e$ must be consistent with those of $m$. For example, a clause $l(v_2) = \text{acknowledge} \wedge s(v_2) = r(v_1) \wedge r(v_2) = s(v_1)$ in the above formula means that there must be an event fixed by $v_2$ such that $l_E(v_2)$ is acknowledge and, $s_E(v_2)$ and $r_E(v_2)$ are $r_E(v_1)$ and $s_E(v_1)$, respectively. In other words, it asserts that a philosopher receiving a message acknowledge from a fork $f$ must be who sent the message request to $f$.

The function *formula* generates a temporal formula capturing the given *order* of messages: the resulting formulae of *formula* can be characterized by the following context free grammar.

$$\begin{aligned}\varphi &:= \Diamond v.\big(\phi \wedge \odot\varphi\big)\\ &\ \ \mid\ \ \Diamond v.\phi\end{aligned}$$

where $\phi$ is a clause generated by the function *node*. Recursively $\varphi$ asserts that eventually ($\Diamond$) an event fixed by $v$ must make $\phi$ hold and next ($\wedge\odot$) the following $\varphi$ must hold. It must be noted that the eventuality operator $\Diamond$ allows the arbitrary distance between related events.

Finally, the function *sem* asserts that at least one of temporal formulae generated from *formula* must be true. It means that there must be at least one sequence of events that constitute one of possible sequences of messages denoted by $seq(M, <)$. In that case, for a characterizing formula $f$ of a sequence diagram $S$, the pair of a given trace $\sigma$ and empty environment $\varnothing$ satisfy $f$. Then, the satisfaction relation $\models$ of HDTL can characterize the environment $\mathcal{E}$ at the point that the validity of $f$ is revealed. Note that every events in $\mathcal{E}$ satisfy the constraint on roles and message labels as well as ordering constraints. In addition, the number of variables in $f$ is same to that of messages in $S$. Thus, these events form the instance of the sequence diagram $S$.

## 4. Extension on tableau method

Originally, the goal of the extension on tableau method is to check the validity of HDTL formulae [2]. As noted earlier, this capability allows us to identify instances of a sequence diagram over a trace: we define the semantics of sequence diagrams such that, for a sequence diagram $S$, a resulting HDTL formula $f$ holds if and only if there exists a sequence of events to form the instance of $S$. In this case, the events accumulated in the environment form the instance of $S$.

Before explaining the extension, let us examine the idea of tableau method for the standard linear-time temporal logic briefly. The key idea behind tableau method is that any temporal formula can be split into two conditions: a non-temporal condition on the current state and a temporal condition on the rest states, called a *present* condition and a *future* condition, respectively. For example, a formula $\Box f$ can be split into $f'$ on a current state and $\bigcirc\Box f''$ on rest states. The conjunction of two conditions, say $f' \wedge \bigcirc\Box f''$, is equivalent to the original formula $\Box f$. Because the present formulae contain no temporal operator, it is possible to check their validity without examining the proceeding events.

Based on the above idea, tableau method defines a set of *splitting rules* that normalize $f$ into the form $\bigvee(f'_i \wedge \bigotimes f''_i)$ where $\bigotimes$ is either $\bigcirc$ or $\bigodot$. Note that each $f'_i$ denotes a present condition while each $f''_i$ denotes a future condition. Applying the splitting rules recursively, we can construct a finite directed graph, called *tableau*, where its nodes and edges denote future conditions and present conditions, respectively: suppose that there be a node $n$ denoting $f$. From $f$ we get a normalized formula $\bigvee(f'_i \wedge \bigotimes f''_i)$ by applying the splitting rules. Then, for each disjunct $f'_n \wedge \bigotimes f''_n$ in $\bigvee(f'_i \wedge \bigotimes f''_i)$, we add a node $n'$ denoting $f''_n$ and connect $n'$ to $n$ by an edge denoting $f'_n$. Since each application of the splitting rules strictly decreases the size of formulae, the construction always terminates.

Table 1
Splitting rules for tableau construction

| | | | |
|---|---|---|---|
| $[r\wedge]$ | $(f_1 \wedge f_2)\mathcal{E}$ | $\rightarrow$ | $\{f_1\mathcal{E}, f_2\mathcal{E}\}$ |
| $[r\vee]$ | $(f_1 \vee f_2)\mathcal{E}$ | $\rightarrow$ | $\{f_1\mathcal{E}\}, \{f_2\mathcal{E}\}$ |
| $[r\Rightarrow]$ | $(f_1 \Rightarrow f_2)\mathcal{E}$ | $\rightarrow$ | $\{(\neg f_1)\mathcal{E}\}, \{f_2\mathcal{E}\}$ |
| $[r\Box]$ | $(\Box f)\mathcal{E}$ | $\rightarrow$ | $\{f\mathcal{E}, (\bigcirc\Box f)\mathcal{E}\}$ |
| $[r\Diamond]$ | $(\Diamond f)\mathcal{E}$ | $\rightarrow$ | $\{f\mathcal{E}\}, \{(\bigodot\Diamond f)\mathcal{E}\}$ |
| $[r\text{frz}]$ | $(x.f)\mathcal{E}$ | $\rightarrow$ | $\{f\mathcal{E}[x := \sigma_0]\}$ |

The meaning of tableau can be explained inductively in the following manner: at a node $n$ we expect the formula $f$ denoted by $n$ to hold with respect to a given trace $\sigma$. This expectation will hold when $f'$ denoted by one $e$ of $n$'s outgoing edges be true with respect to $\sigma_0$, and if $f''$ denoted by the target node $n'$ of $e$ hold with respect to $\sigma^1$. By repeating the analysis on $f''$ denoted by $n'$ inductively, we can determine the validity of $f$ over the trace $\sigma$.

The tableau method can be extended to support HDTL as shown in Table 1. It should be noted that the rule $[r\text{frz}]$ requires an event $\sigma_i$ to discharge a quantifier. That is, it is impossible to construct a tableau for an HDTL formula. Therefore, our analysis should be conducted with one specific trace. This motivates us to design a deferred representation called a *flow tree*. Flow tree is constructed with given a flow tree $\mathcal{T}$ consists of a set of location $\mathcal{L}$, a set of transition $\mathcal{T}$, a flow relation $\mathcal{F} \subseteq (\mathcal{L} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{L})$, and a labeling function $\mathcal{M}$ that associates locations and transitions with HDTL formulae. When $\mathcal{F}(l, t)$ and $\mathcal{F}(t, l')$, a tuple $\langle t, l' \rangle$ is said to be the *branch* of a location $l$.

Initially, a flow tree $\mathcal{T}$ consists of only a root location $l$ denoting an original HDTL formula $f$. To check the validity of $f$ over a trace $\sigma$, we expand the branches $\langle t_i, l'_i \rangle$ of $l$ such that $\mathcal{M}(l) \equiv \bigvee(\mathcal{M}(t_i) \wedge \bigotimes \mathcal{M}(l'_i))$ where $\bigotimes$ is either $\bigcirc$ or $\bigodot$ by applying the splitting rules in Table 1. Then, for each branch $\langle t, l' \rangle$, we check the validity of $t$ over $\sigma_0$ and repeat the same analysis on $l'$ over $\sigma^1$ when $t$ holds. During analysis, we keep the environment for each location $l$. Note that more than one branch can remain because flow trees are nondeterministic in the sense of automata theory. These lowest locations are referred to as *leaf locations*.

Note that leaf locations are enough to determine the acceptance of the trace. This observation makes it possible to reduce the effort of maintaining flow trees: we only maintain leaf locations and remove other locations and transitions. For example, the trace in Fig. 2 results in three leaf locations denoting the following formula and environment tuples (1), (2), and (3):

$$\begin{aligned}&\Big\langle \mathsf{true}, \Big\{\big(v_1, \langle \mathsf{request}, \mathsf{p\#1}, \mathsf{f\#1}\rangle\big),\\ &\qquad \big(v_2, \langle \mathsf{acknowledge}, \mathsf{f\#1}, \mathsf{p\#1}\rangle\big),\\ &\qquad \big(v_3, \langle \mathsf{release}, \mathsf{p\#1}, \mathsf{f\#1}\rangle\big)\Big\}\Big\rangle, \qquad (1)\end{aligned}$$

$$\begin{aligned}&\Big\langle \mathsf{true}, \Big\{\big(v_1, \langle \mathsf{request}, \mathsf{p\#2}, \mathsf{f\#2}\rangle\big),\\ &\qquad \big(v_2, \langle \mathsf{acknowledge}, \mathsf{f\#2}, \mathsf{p\#2}\rangle\big),\\ &\qquad \big(v_3, \langle \mathsf{release}, \mathsf{p\#2}, \mathsf{f\#2}\rangle\big)\Big\}\Big\rangle, \qquad (2)\end{aligned}$$

$$\big\langle F_1, \{\}\big\rangle. \qquad (3)$$

In this case, the first two tuples, (1) and (2), indicate that two instances of the sequence diagram have occurred, and show which events collectively constitute the instances. The last simply says that new instances of the sequence diagram can always begin. In this way, we can identify instances of a sequence diagram over a trace.

## 5. Concluding remarks

In this paper, we presented the formal semantics of sequence diagrams that facilitates the generic feature as well as automatic analysis. Our semantics is based on a variant of temporal logic, named HDTL, employing the freeze quantifier. This approach allows us to identify the instance of a sequence diagram over a trace. To do this, we developed a deferred representation of infinite tree, named a flow tree.

This capability raises an interesting question: when an event participates in more than one instance of sequence diagrams, is it an error or not? Since a sequence diagram is assumed to capture a set of coherently related events, the above case looks erroneous. However, in some cases, for a given event $e$, the proposed semantics can not uniquely identify the causal event. To resolve this problem, we think elaborate survey for the usage of sequence diagrams in practice is required.

## References

[1] O. Biberstein, D. Buchs, N. Guelfi, Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism, in: Advances in Petri Nets on Object-Orientation, Springer, Berlin, 1997.
[2] S.M. Cho, H.H. Kim, S.D. Cha, D.H. Bae, Specification and validation of dynamic systems using temporal logic, IEE Proc. Software 148 (4) (2001).
[3] L.K. Dillon, Y.S. Ramakrishna, Generating Oracles from your favorite temporal logic specifications, in: Proc. 4th ACM SIGSOFT Symp. Foundations of Software Engineering, San Francisco, 1996, pp. 106–117.
[4] P. Graubmann, E. Rudolph, J. Grabowski, Towards a Petri net based semantics definition for message sequence charts, in: Proc. SDL'93: Using Objects, Elsevier, Amsterdam, 1993.
[5] T.A. Henzinger, Half-order modal logic: How to prove real-time properties, in: Proceedings of the 9th Annual Symposium on Principles of Distributed Computing, 1990.
[6] P.B. Ladkin, S. Leue, Interpreting message flow graphs, Formal Aspects Comput. 3 (1994).
[7] J. De Man, Towards a formal semantics of message sequence charts, in: Proc. SDL'93: Using Objects, Elsevier, Amsterdam, 1993.