韓國 컴퓨터情報學會 論文誌
第11卷 第6號, 2006. 12

2006-11-6-1-6

Live Sequence Chart 명세언어의 의미론적 정의

이 은 영*

Defining Semantics of Live Sequence Chart Specification

Eun-Young Lee*

요 약

사용자와 상호작용을 하는 복잡한 기능을 가진 소프트웨어 시스템을 구현하는데 있어서, 사용자의 요구를 분석하고 이를 개발되는 시스템에 제대로 반영하는 매우 중요한 일이다. 따라서 사용자의 필요를 빠르고 정확하게 이해하는 것이 성공적인 소프트웨어 시스템을 개발하는 중요한 열쇠가 된다. 여러 가지의 요구 명세 언어 중에서도 UML의 Sequence Diagram으로 알려져 있는 Message Sequence Charts (MSC)는 시나리오 개념을 가장 잘 표현하고 있는 언어라고 할 수 있다. Live Sequence Charts (LSC)는 MSC의 확장된 형태로 메시지 추상화와 시나리오의 모드 설정이 가능하다는 특징을 가지고 있다. 본 논문에서는 기존의 연구에서는 다루어지지 않았던 LSC 명세언어의 주요 생성자들을 모두 포괄할 수 있는 LSC 명세언어의 의미론을 새롭게 정의하고 논의하였다. 본 논문에서 정의된 의미론은 기존의 방법들과 비교했을 때 훨씬 넓은 범위의 LSC 명세언어를 포괄하고 있으며, 그동안 정형화되지 않았던 기존의 LSC 명세언어의 내용을 가장 정확하게 표현하고 있다는 장점을 가지고 있다.

Abstract

While developing a complex reactive software system, it is very important to analyze the user requirement and reflect it to the developed system. Therefore understanding the need of users precisely and promptly is the key to the successful software system development. Among several requirement specification languages, message sequence charts (MSCs), also known as sequence diagrams in UML are the most widely used scenario notation. Live Sequence Charts (LSCs) are a variant of MSCs, characterized by its message abstraction facility and the modality of scenarios. In this paper, I define the formal semantics of LSC specification including the essential language constructs such as pre-charts, variables, assignment and conditions. The range of the formalized LSC language has been broadened, and the scope of the formalized semantics is much closer to the complete LSC specification.

▸ Keyword : UML, Message Sequence Charts, Scenario-based Language, Formal Specification, Language Semantics

---

• 제1저자 : 이은영
• 접수일 : 2006.11.01, 심사일 : 2006.12.01, 심사완료일 : 2006. 12.20
* 동덕여자대학교 정보과학대학 컴퓨터학과 교수
※ 본 연구는 2005년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것임.

Ⅰ. Introduction

While developing a large reactive software system, it is very important to analyze the user's requirement and refelct it to the developed system. Therefore understanding the need of users precisely and promptly is the key to successful software development. Much research has been recently done to make the communication between users and system developers easier, resulting in several requirement specification languages [1],[2],[3],[4].

The Unified Modeling Language (UML) [5],[6],[7] is the leading standard for specifying object oriented software systems. Among various notations in UML, sequence diagrams are models that describe how groups of objects collaborate in some behavior. Typically, a sequence diagram captures the behavior of a single use case. The diagram shows a number of example objects and the messages that are passed between these objects within the use case. The sequence diagrams of UML, in fact, are a variant of classical message sequence charts (MSCs) [8]. MSCs are suffered from two serious shortcoming: they do not provide message abstraction and they do not support the modality of charts.

Live Sequence Chart (LSC) specification has extended MSCs, overcoming those shortcomings. The LSC specification is designed by Harel and Damm in [9],[10]. The LSC specification is a visual formalism based on specifying the various kinds of scenarios of the system - including those that are mandatory, those that are allowed but not mandatory, or that are forbidden. MSCs can be viewed as a subset of LSCs, the existential charts.

Visual formalism such as UML, MSCs and LSCs is a very strong tool to visualize the design of a system, specifying the behavior of a target system, or describing user requirement. Visual formalism, however, is usually suffered from the lack of formally defined semantics for its own language. The LSC specification is not the exception for this deficiency.

In this paper, the formal semantics of LSC specification is defined including the semantics of the essential language constructs such as pre-charts, variables, assignment, and conditions. Those language constructs are excluded in the previous work for simplicity or for other reasons, but to enjoy expressiveness of LSC specification in a full strength, those constructs should be used at designing time and at verification time. The range of formalized language is broader than the previous work, and much closer to the complete LSC semantics.

I discuss the related work on visual formalism and formalizing the semantics of scenario-based languages in Section 2. In Section 3, we provide a basic idea of the LSC specification, focusing on its graphical notation. Section 4 is devoted to explaining the core concepts of the LSC specification. Section 5 describes the basic LSC constructs formal semantics and how I extend the formal semantics to express the other core LSC constructs in a formal way. The benefits and the future work of the approach are discussed in Section 6.

Ⅱ. Related Work

Message Sequence Charts are widely used to describe scenarios [11].

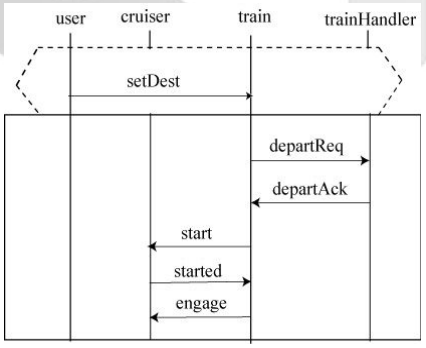

Figure 1. LSC Universal Chart
그림 1. LSC 범용 차트

They are easy to understand with highlighting actor interactions. Moreover, the MSC standard [8] provides a structuring language, High-Level Message Sequence Charts (HMSC), that makes it possible to compose scenario through sequence, iteration, concurrency or choice.

In spite of all its advantages, MSCs suffer two serious shortcomings: they do not allow message abstraction, and they do not explicitly mention the status of each scenario.

Message abstraction makes it possible to state that only the messages appearing in the chart are relevant, omitting all the other irrelevant messages. In Figure 1, the scenario specifies only 4 participating objects and their exchanging messages. Assuming there exists a clock for synchronization, and the clock sends messages about the current time to the objects in the system. The scenario in Figure 1 will remain silent when any of the object in the scenario (for example, train) asks the clock for the current time. The scenario does not disallow this kind of requests, and those requests never cause any violation of the scenario.

The status of the behavior described by the MSC is not very clear: It is hard to tell whether a scenario is a simple example (the system may sometimes behave like that) or a universal rule (the system must behave as specified, if a given condition is met).

The ability to abstract away irrelevant messages and to specify the status of each scenario is a characteristic of Live Sequence Charts (LSC). LSCs are a good alternative for overcoming the shortcoming of MSCs. However, the LSC specification still lacks in formally defined semantics for its essential constructs.

Harel and Kugler proposed a way of synthesizing object systems from LSC scenarios [12]. But the language they had used for synthesis is quite different from the complete LSC specification in [8]. For example, Harel and Kugler excluded pre-charts from their research, which are considered as one of most prominent features of the LSC language. Their language allows only simple message exchange with sacrificing all the other enriched language constructs of LSCs, therefore the proposed algorithm cannot handle anything more than messages.

Bontemps and Heymans adopts another variant of LSC specification called High-level LSCs (HLSCs) for their synthesis algorithm [13]. HLSCs are a variant of Harel and Kugler's restricted LSCs, extended with composition operators. Their HLSC specification consists of three layers titled Basic Charts (BCs), Iterative Charts (ICs), and Live Sequence Charts (LSCs). However, Bontemps and Heymans' HLSC specification cannot be considered as a subset of the current LSC specification because it has quite different semantics and language constructs from the complete LSC specification in [10]. The HLSC specification can be categorized as a new specification language, adding the chart modality to an existing scenario-based language. Bontemps and Heymans' HLSC specification and their synthesis algorithm do not support the enriched language constructs of LSC specification, suffering from the same shortcoming as Harel and Kugler's.

III. Live Sequence Charts (LSC)

In this section, I introduce a train control system, which will be used as an example to explain the main ideas of the LSC specification. This system, in fact, is a famous example which has been used to describe the ideas and the semantics of LSC for a long time [10],[12],[14].

Suppose an automated train control system which controls the departure and arrival of trains between train stations. There are several required behaviors for the automated trains (or the train control system) in order for the safe transport of passengers to be guaranteed. Those requirements can be specified using LSCs.

Figure 1 describes the requirement which a train departing from a train station must follow. The objects participating in this scenario are **user, cruiser, train, trainHandler**. The chart describes the message communication between the objects, with time propagating from top to bottom. An LSC consists of two parts: a prechart and a main chart. Any message sequence in the main chart can happen if and only if the message sequence in the pre-chart occurs before it. The pre-chart is shown in the upper part of the chart in dashed line-style, it can be considered as a precondition that must be satisfied before any message sequence occurs in the main chart. The chart of Figure 1 is called an universal chart. If a system satisfies an LSC universal chart, and if the pre-chart of the universal chart is satisfied (i. e. the message sequence

of the pre-chart has occurred), every run of the system must satisfy the main part of the universal chart.

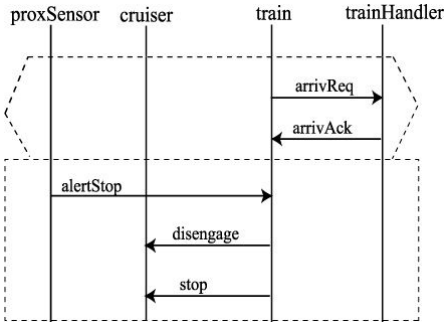

Figure 2. LSC Existential Chart
그림 2. LSC 용례 차트

Whenever the train receives the message **setDest** from the user, the sequence of messages in the chart should occur in the following order: the train sends a departure request **departReq** to the trainHandler, which sends back a departure acknowledgement **departAck** to the train. The train then sends a **start** message to the cruiser in order to activate the engine, and the cruiser responds by sending the started message to the train. Finally, the train sends an **engage** message to the cruiser, and then the train can depart from the train station.

Figure 2 and Figure 3 are existential charts, depicted by dashed borderlines. Those existential charts describe two possible scenarios in which a train approaches a train station: stop at the train station or pass through the station without stopping.

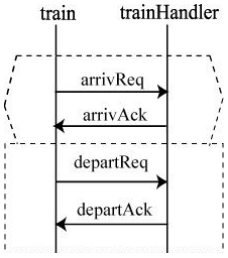

Figure 3. LSC Chart
그림 3. LSC 차트 예제

Any system satisfying an existential chart must has *at least one system run,* in which if the pre-chart of the existential chart holds, then the main part is satisfied. Two charts show what could happen after the train sends an arriving request **arrivReq** to the trainHandler, and gets an acknowledge **arrivAck** from the trainHandler. If the train is going to stop at the station, the message sequence depicted at the main part of Figure 2 will occur. If the train is going to pass through the station, the sequence of messages in Figure 3 will occur. It is impossible for every system run of a system to satisfy both of two existential charts. An existential chart, however, are considered to be satisfied if there exists at least one system run corresponding to the chart. In an iterative development of LSC specifications, such existential charts may be considered underspecified, and can be refined and be transformed into universal charts at the later part of development. Existential charts are also used as testing scenarios after a system is implemented with LSC specifications. This explains the modality of scenarios which the LSC specification provides.

IV. LSC Concepts

The complete semantics of the LSC language is defined in [10], and I explain the basic definitions and concepts of the language with the automated train control system of the previous section. The formal semantics of LSC language is given in Section 5.

Figure 4 is the same to the chart shown in Figure 1, but it is labeled with the **locations** of the objects. The set of locations for the chart is as follows:

{<*user*,0>,<*user*,1>,<*cruiser*,0>,<*cruiser*,1>,<*cruiser*,2>, <*cruiser*,3>,<*train*,0>,<*train*,1>,<*train*,2>,<*train*,3>, $\langle train,4\rangle,\langle train,5\rangle,\langle train,6\rangle,\langle trainHandler,0\rangle$, <*trainHandler*,1>, <*trainHandler*,2>}

The chart defines a partial order $<_m$ on locations. The requirement for order along an instance line implies, for example, $\langle train,0\rangle <_m \langle train,1\rangle$. The order is also caused by sending and receiving messages, for example, $\langle train,2\rangle <_m \langle trainHandler,1\rangle$. The partial order $<_m$ is transitive, so from the transitivity, another order $\langle train,0\rangle <_m \langle trainHandler,1\rangle$ can be induced.

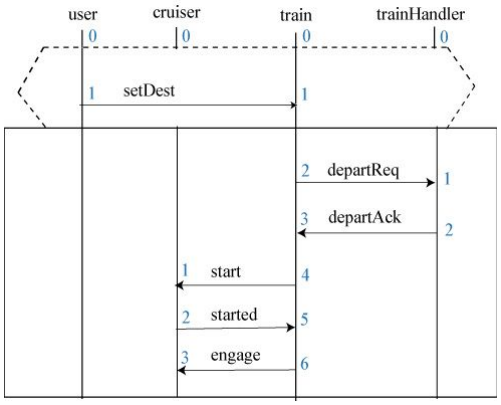

Figure 4. LSC Chart with Locations
그림 4. 객체별 위치가 표시된 LSC 차트

One of the basic concepts of LSC specification is the notation of a **cut**. A slice in a chart is defined as a tuple consisting of one location of each participating object. A set of cuts for an LSC chart is a subset of slices in the chart. Intuitively a cut through a chart represents the progress each instance has made in the scenario. Not every slice is a cut. For example, the slice (<*cruiser*, 1>, <*train*, 3>, <*trainHandler*, 2>) is not a cut. Before the cruiser receives a message **start** from the train, the train must send the message. It requires the train should have already reached the location <*train*, 4>.

The cuts for the chart in Figure 5 are:

{(<*user*,0>, <*cruiser*,0>,<*train*,0>,<*trainHandler*,0>),
(<*user*,1>, <*cruiser*,0>,<*train*,0>,<*trainHandler*,0>),
(<*user*,1>, <*cruiser*,0>,<*train*,1>,<*trainHandler*,0>),
(<*user*,1>, <*cruiser*,0>,<*train*,2>,<*trainHandler*,0>),
(<*user*,1>, <*cruiser*,0>,<*train*,2>,<*trainHandler*,1>),
(<*user*,1>, <*cruiser*,0>,<*train*,2>,<*trainHandler*,2>),
(<*user*,1>, <*cruiser*,0>,<*train*,3>,<*trainHandler*,2>),
(<*user*,1>, <*cruiser*,0>,<*train*,4>,<*trainHandler*,2>),
(<*user*,1>, <*cruiser*,1>,<*train*,4>,<*trainHandler*,2>),
(<*user*,1>, <*cruiser*,2>,<*train*,4>,<*trainHandler*,2>),
(<*user*,1>, <*cruiser*,2>,<*train*,5>,<*trainHandler*,2>),
(<*user*,1>, <*cruiser*,2>,<*train*,6>,<*trainHandler*,2>),
(<*user*,1>, <*cruiser*,3>,<*train*,6>,<*trainHandler*,2>)}

The **run** of a chart consists of a sequence of cuts arranged by the order in which the messages are sent. The **trace** of a run shows the sequence of messages which are sent and received during the execution of the run. A trace is an ordered set of pairs such that each of the pairs consists of the instance which initiates a message, and the sent message. The trace of the chart in Figure 5 is:

{(*user*, *train.setDest*),
(*train*, *trainHandler.departReq*),
(*trainHandler*, *train.departAck*),
(*train*, *cruiser.start*), (*cruiser*, *train.started*),
(*train*, *cruiser.engage*)}

As part of the "liveness" property, the LSC language enables forcing progress along an instance line. Each location, or each message can have a temperature **cold** or **hot**. The temperature of a location is depicted by dashed or solid segments of the instance line. A run must continue down solid lines, while it may continue down dashed lines. The temperature of a message is graphically denoted by dashed or solid arrows between instance lines. Sending a hot message means the message delivery must happen. That is, the message will eventually get to the receiver of the message. Unlike hot messages, the delivery of cold messages are not guaranteed, therefore cold messages can be missing during delivery. The system satisfying an LSC specification should take this into consideration. By the definition of LSC language, all the locations at the last cut in a run should be cold.

V. Formal Semantics of LSC Language

This section defines the formal semantics of LSC language in [10]. The complete LSC language is quite complex and versatile, and this expressiveness makes LSC language a useful tool to depict scenarios for a system development. Few research has been conducted for formalizing the semantics of LSC language, and only restricted and simplified version of the language was used for the recent researches [12],[13]. In the simplified version, they did not consider pre-charts, variables, or conditions. Although using restricted version made the research simple and easily understandable, it significantly

diminishes the expressive power of LSC language. I define the semantics of LSC language which include those essential language constructs (modality, pre-charts, variables, assignment, and conditions) as well as other language constructs which were already tackled in the restricted version of the language. It is assumed that all messages are synchronous and that there are no failures in the system.

5.1 Basic Definitions

A **chart** of LSC specification is a tuple of its mode, a set of messages and a set of conditions. The mode of a chart specifies whether a chart is universal or existential.

$$Chart = Mode \times Set(Message) \times Set(Condition)$$
$$Mode = \{ Universal, Existential \}$$

The messages and the conditions are the contents depicted in the chart, and their formal semantics will be discussed later in this section.

The system related to an LSC specification is usually composed of a set of objects, and the objects are called **instances**. They are entities interacting with each other in a chart, and graphically denoted as vertical lines in a chart. LSCs specify the behavior of a system in terms of the message communication between the instances in the system. For a chart m, $inst(m)$ is the set of all instance-identifiers referred to in the chart. With each instance i, a finite number of locations are assigned $loc(m,i) \subseteq \{0, \ldots, l_{max}(i)\}$. The set of all locations of a chart m is:

$$loc(m) = \{ <i,l> \mid i \in inst(m) \wedge l \in loc(m,i) \}$$

We will call $loc(m)$ **Locations**. Each location in the chart m has been associated with its temperature as explained in Section 4. A mapping *temp* can be defined for the temperature of a location as follows:

$$temp(m): loc(m) \rightarrow Temp$$

where the set $Temp = \{ hot, cold \}$.

An object usually maintains variables which keep its internal state. Some variables are visible to other objects while others are kept in private. The instances of an LSC chart, as an abstraction of objects, maintain variables for the same purpose. We call a visible variable of an instance **Property**. Let $properties(m)$ be the set of all property-identifiers in chart m. Each variable is associated with its domain, i. e. the range of values the variable could have. Let $domain(p)$ be the domain set of the property p.

When a system is specified using the LSC language before development, the resulting LSC specification usually ends up with more than one charts. In order to analyze the charts or synthesize a state machine out of charts, it is needed to keep track of all the variables of given charts. We define a mapping which returns all visible variables:

$$properties_{global}: Set(Chart) \rightarrow Set(\text{Pr}operty)$$

An **event** appearing in chart m is a tuple of five parts:

$$Event = loc(m) \times \Sigma \times loc(m) \times \text{Pr}operty \times Values$$

where $(<i,l>, \sigma, <i',l'>, p, v)$ corresponds to instance i at location l, sending a string σ to instance i' at location l'. The string σ is an identifier standing for the sent message, and the message changes the value of property p into the value v. The property p must be the variable of instance i', and must be visible to other instances. Let $event(m)$ be the set of events appearing in chart m.

Finally, the **messages** in chart m are pairs:

$$Message(m) = Event(m) \times Temp$$

Let $msg(m)$ be the set of messages appearing in chart m. Each location can appear in at most one message in the chart, and each message is associated with its temperature. The function *temp* is defined polymorphically for mapping a message to its temperature as follows:

$$temp(m): Message(m) \rightarrow Temp$$

5.2 Cuts

A **cut** c is specified by the locations, one for each instance in the chart:

$$c = \{ <i_{1,}l_1>, <i_{2,}l_2>, ..., <i_{n,}l_n> \}$$

Harel [10] defined the **preset** of a location $<i,l>$ containing all elements in the domain of a chart smaller than $<i,l>$:

$$preset(<i,l>) \equiv \{ <i',l'> \in loc(m) \mid <i',l'> \leq_m <i,l> \}$$

where the partial order $\leq_m$ is defined as follows:

- order along an instance line:

$$\forall \langle i,l \rangle \in loc(m),\ l < l_{\max} \Rightarrow \langle i,l \rangle \leq_m \langle i,l+1 \rangle$$

- order induced from message sending:

$$\forall m \in msg(m).\ m = (\langle i,l \rangle, \sigma, \langle i',l' \rangle, *, *, hot) \Rightarrow \langle i,l \rangle \leq_m \langle i',l' \rangle$$

- messages blocking sender until receipt:

$$\forall m \in msg(m).\ m = (\langle i,l \rangle, \sigma, \langle i',l' \rangle, *, *, hot) \Rightarrow \langle i',l' \rangle \leq_m \langle i,l+1 \rangle$$

As we mentioned before, all the slices in a chart are not cuts. A cut c through chart m is a set of locations such that for every location $<i,l>$ in the cut c, the pre-set of c does not include any location $<j,l'>$ satisfying $<j,l_j> \leq_m <j,l'>$ for some location $<j,l_j>$ in c. The initial cut of a chart denotes a cut in which all the instances are at Location 0. The final cut of a chart is a cut in which the location of each instance in the chart is its last location. Let *initialCut(m)* return the initial cut of chart m, and let *finalCut(m)* return the final cut of chart m.

For chart m, some $1 \leq j \leq n$ and cuts c, c', with

$$c = (<i_{1,}l_1>, <i_{2,}l_2>, ..., <i_{n,}l_n>)$$

$$c' = (<i_{1,}l_1'>, <i_{2,}l_2'>, ..., <i_{n,}l_n'>),$$

$suc_m(c, <i_j,l_j>, c')$ holds if c and c' are both cuts, and if $l'_j = l_j + 1 \wedge \forall k \neq j.\ l'_k = l_k$.

We denote the **cutAdvance** of a chart m, a cut c and a message $(<i,l_i>, \sigma, <j,l_j>, *, *, hot)$ returning the next cut c', which can be reached from the cut c when the message $(<i,l_i>, \sigma, <j,l_j>, *, *, hot)$ is received in chart m. An instance in an LSC chart can send messages to itself (called self messages), and the next cut at receiving a self message is different from the next cut at receiving a general message. Within a system satisfying some LSC specifications, the charts can be activated more than once. That means, if the system run reaches at the final cut c of chart m, then the next cut of c' will be the initial cut of chart m. The chart m will get activated again if the first message in the pre-chart of m occurs.

The mapping *cutAdvance* is defined as follows:

- self messages: $i = j \wedge l_i = l_j$

$$cutAdvance(m, c, (\langle i,l_i \rangle, \sigma, \langle j,l_j \rangle, *, *, hot)) \equiv \begin{cases} c' & : suc_m(c, \langle j,l_j \rangle, c') \\ initialCut(m) & : \exists c'.\, suc_m(c, \langle i,l_i \rangle, c') \wedge c' = finalCut(m) \end{cases}$$

- general message: $i \neq j$

$$cutAdvance(m, c, (\langle i,l_i \rangle, \sigma, \langle j,l_j \rangle, *, *, hot)) \equiv \begin{cases} c' & : \exists c''.\, suc_m(c, \langle i,l_i \rangle, c'') \wedge \\ & \quad suc_m(c'', \langle j,l_j \rangle, c') \\ initialCut(m) & : \exists c'.\, \exists c''.\, suc_m(c, \langle i,l_i \rangle, c'') \wedge \\ & \quad suc_m(c'', \langle j,l_j \rangle, c') \wedge \\ & \quad c' = finalCut(m) \end{cases}$$

5.3 Environments

The formal semantics explained in this subsection are closely related to the LSC language constructs for handling variables, assignment, and conditions. First, I introduce **Context** that formally describes the semantics for variables, and then explain how to express conditions in terms of the semantics of LSC language.

The **valuation** of a variable is a pair of variable name and its value. For example, *(x, 3)* and *(trainHandler:depRequested, 0)* are valuations.

Context is denoted as a set of valuations. A context cannot be a multi-set, i. e. a context contains only one valuation for each variable. The set *domain(p)* is the finite set of possible values which the variable p can have. For example, *domain(p) = {0, 1}* if the variable p is of boolean type. The mapping *contexts(m)* returns a set of contexts, containing all possible contexts of chart m. It is defined as:

$$contexts(m) \equiv \left\{ \{(x_1,v_1),...,(x_n,v_n)\} \mid \begin{array}{l} \text{for } i = 1,...,n,\ x_i \in properties(m) \wedge \\ v_i \in domain(p), \end{array} \right\}$$

For a context Γ, we define a new context Γ', an

***x*-variant** of Γ. Intuitively an x-variant of Γ is a context which has a different valuation for the variable x, but not anything else.

$$variant(\Gamma,\Gamma',x) \equiv \begin{array}{l}\exists v.\exists v'.(x,v)\in\Gamma\wedge(x,v')\in\Gamma'\wedge\\ (v\neq v')\wedge(\forall y.\forall v.(y\neq x)\Rightarrow\\ ((y,v)\in\Gamma\Leftrightarrow(y,v)\in\Gamma'))\end{array}$$

Two context are equivalent in term of variable x, if they agree with the value of x. The mapping ***x*-equivalence** is:

$$=_x(\Gamma,\Gamma')\equiv\exists v.((x,v)\in\Gamma)\wedge((x,v)\in\Gamma')$$

A context Γ can be extended with a variable x unless the context Γ already contains the valuation of the variable x. The ***x*-extension** of a context Γ is a set of contexts defined as follows:

$$extend(\Gamma,x)\equiv\left\{\Gamma\cup\{(x,v)\}\mid\begin{array}{l}(\neg\exists v'.(x,v')\in\Gamma)\wedge\\ v\in domain(x)\end{array}\right\}$$

5.4 Conditions and States

A **condition** in a chart is denoted as a tuple of a cut before the condition, a cut after the condition, and its conditional expression. The LSC specification allows only propositional formulae in a conditional expression.

$$Condition = Cut\times Cut\times Formula$$

Each cut in an LSC chart is associated with a context of visible properties. **States** in an LSC chart are pairs of a cut and a context. The context could be an empty set if there is no visible properties in the chart.

$$State = Cut\times Context$$

The mapping *stateEquiv* returns a set of states, equivalent in term of a cut c in chart m. All the states of the set are for the cut c, but they have different contexts for visible properties.

$$stateEquiv(m,c)\equiv\{(c,\Gamma)\mid\Gamma\in contexts(m)\}$$

VI. Conclusion

Visual formalism with graphical notations is very useful for analyzing user requirements, and for communication between clients and developers. UML is one of the most successful requirement specification languages, and sequence diagrams are one component of UML used for describing the behavior of a system. The LSC specification is an extended variant of MSCs (a predecessor of sequence diagrams), giving more expressiveness and correctness to scenarios. The LSC specification, however, has been suffered from the lack of formal semantics like other requirement specification languages.

In this paper, the formal semantics of LSC specification has been defined including the semantics of the essential language constructs such as pre-charts, variables, and conditions, which were excluded from the previous research on LSCs. The range of the formalized LSC language has been broadened, and the scope of the formalized semantics is much closer to the complete LSC specification.

Automatic synthesis of software system out of graphical requirement specification is another interesting research topic in requirement analysis. I think the formal semantics in this paper will be of great help for finding a way of synthesizing a system from enriched scenario based languages like LSCs.

References

[1] D. Harel. Statecharts: a visual formalism for complex systems. Science of Computer Programming, 8:231-274, 1987.

[2] J. Coplien and D. Schmidt. Pattern Languages of Program Design. Addison-Wesley, 1995.

[3] P. Kruchten. The Rational Unified Process: An Introduction. Addison-Wesley, 1999.

[4] G. Schneider and J. Winters. Applying Use Cases: A Practical Guide. Addison-Wesley, 1998.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley Professional, 1st edition, September 1998.

[6] UML. Documentation of the Unified Modeling Langauge (UML). Available from the Object Management Group (OMG), http://www.omg.org

[7] UML Distilled. M. Fowler and K. Scott. 2nd Edition. Addison-Wesley. 1999.

[8] ITU-TS recommendations Z.120: Message sequence chart (MSC). http://www.itu.int/, 1996.

[9] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. Formal Methods in System Design, 19(1):45-80, 2001.

[10] D. Harel and R. Marelly. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play Engine. Springer, 2003.

[11] D. Amyot and A. Eberlein. "An Evaluation of Scenario Notations for Telecommunication Systems Development." In Proc. of 9th Int. Conference on Telecommunication Systems (9ICTS), Dallas, USA, March 2001.

[12] D. Harel and H. Kugler. Synthesizing state-based object systems from lsc specifications. International Journal of Foundations of Computer Science, 13(1):5-51, February 2002.

[13] Y. Bontemps and P. Heymans. Turning high-level live sequence charts into automata. In Proc. of Scenarios and State-Machines: models, algorithms and tools workshop of the 24th Int. Conf. on Software Engineering, Orlando, FL, May 2002.

[14] D. Harel and E. Gery, "Executable Object Modeling with Statecharts", IEEE Computer (July 1997), pages 31-42.

[15] D. I. Cohen. Introduction to Computer Theory. Wiley, 1996.

[16] J. Drissi and G. v. Bochmann. Submodule construction tool. In Proc. of International Conference on Computational Intelligence for Modelling, Control and Automation, pages 319-324, Vienne, February 1999. IOS Press.

[17] Z. P. Tao, G. Bochmann, and R. Dssouli. A model and an algorithm of subsystem construction. In Proc. of the Eighth International Conference on Parallel and Distributed Computing Systems, pages 619-622, Orlando, Florida, USA, September 1995.

저 자 소 개

이 은 영

2004년 Ph. D. in Computer Science, Department of Computer Science, Princeton University, U. S. A.

2005년 ~ 현재 : 동덕여자대학교 컴퓨터학과 전임강사

<관심분야> Computer Security, Programming Language, Compiler