ソフトウェア工学 63－2
（1988. 11. 25）

通信サービス記述を指向した図形言語

伊藤 正樹　加藤 順　市川 晴久

NTT ソフトウェア研究所

通信ソフトウェア開発とメンテナンス，特にサービス拡張を指向した図形言語ＧＳＡＬを提案する．ＧＳＡＬはテキスト形式の言語ＳＡＬ（Service Addtion Language, [ICHI86]）を基礎にしており，通信サービスの記述に従来から用いられているメッセージシーケンス図の拡張になっている．ＧＳＡＬの特徴は，読解性の他に，サービスを制御するプロセス間で交わす通信の妥当性がサービス記述時点で保証される点にある．ＧＳＡＬによるサービス記述は，ＳＡＬと相互変換でき，ＳＡＬを経由してＣＣＩＴＴの標準仕様記述言語ＳＤＬに変換される．

A Graphic Language for Telecommunication Services Based on the Message Sequence Chart

Masaki Itoh, June Kato and Haruhisa Ichikawa
NTT Software Laboratories

Abstract

A graphic language, GSAL, which facilitates communications software development and maintenance, especially service enhancement during its life cycle, is presented. GSAL is based on a textual language, *SAL* (Service Addition Language, [ICHI86]) and is an extention of the *message sequence chart* being widely used for telecommunications service description. GSAL features *drawing* capabilities which maintain the consistentcy of the interaction between communicating processes. Descriptions in GSAL can be transformed via SAL into process descriptions in SDL, the CCITT standard specification language.

1 Introduction

The results of *FDT* (Formal Descrition Techniques) use are now coming to fruition in communications software development. The three ISO standards *Estelle* [ESTE88], *ASN.1* [ASN188] and *LOTOS* [LOTO88] have been applied in the area of protocol. *SDL* (Functional Specification and Description Language, [SDL84]), a CCITT standard, has already been widely used in the telecommunication industry. It offers a flow-chart-like graphic version -*SDL/GR*-, which seems to be the main reason for its popularity.

A current trend in FDT use is the growing importance of the *service concept* [VISS85]. The service concept was first presented in [BOCH80] for the extended state transition machine which is the model of Estelle. Also, a more formal concept of the service specifications for LOTOS, which are independent of its implementation, was presented in [BRIN86] and [SCOL86]. Here, the service specification of a system represents how the system interacts with its environment. In layered protocol, for example, the service specification of ith layer protocol represents the interaction with its upper ($i+1$th) layer.

The authors developed a different, SDL-oriented approach to service specification. *SAL* (Service Addition Language, [ICHI86]) captures a service as a tuple of partial behaviours of processes to implement. That is, SAL aims at describing sets of those tuples as requirements for the communicating processes represented in SDL.

GSAL (Graphic SAL), presented in this paper, offers a graphic interface to SAL. As described in Section 2, SAL is compatible with the so called *message sequence chart* which is a widely used graphic expression as is SDL. For this reason, GSAL expressions are an extention of the message sequence charts. GSAL has an important feature that SAL does not. An arrow indicating a message passing in GSAL expressions shows a one-to-one correspondence between the message transfer action of a process and another action to receive it. Therefore, the inter-process communications are validated through the conversion from SAL to GSAL shown in Section 4.

2 Service specification language SAL

2.1 SAL and SDL

Figure 1 illustrates a pair of communicating processes, p and q, in SDL/GR, the graphic version of SDL. Table 1 defines the different box types, though it is not a full set. In Fig. 1, messages to transfer and target process names are described in the input and the output. The description "$q(3: question)$" in the output box means that *process p sends the message "3 : question" to process q.* If the process name is omitted, the box denotes an input or an output from/to its environment. The description ($digit$) in the first input box in Fig. 1 (a) means that *process p receives message digit from its environment.* As is easily seen by tracing the message transfer between p and q, the system specified in Fig. 1 takes one of the following three behaviours:

1. p follows the path p_1 (shown in Fig. 1), and q follows the path q_1. This behaviour is denoted by (p_1, q_1).
2. (p_2, q_2).
3. (p_3, q_3).

In the case of (p_3, q_3), p receives $(address, message)$ from its environment and executes $setadr(q, address)$. Next, p sends message "$8: message$" to q and terminates. Process q sends message $message$ to its environment after receiving "$8: message$" from p.

The SDE [ICHI86] derives the system specifications in SDL from given sets of such tuples as shown above. SAL facilitates describing partial behaviour tuples. Tuples (p_1, q_1), (p_2, q_2) and (p_3, q_3) are described in SAL as shown in Fig. 2. The five lines following the comment *service 1* represent the tuple (p_1, q_1). The next five lines and two lines delimited by '|' represent the tuples (p_2, q_2) and (p_3, q_3), respectively.

These delimited lines of of the service description, which correponds to tuples, are called *multilogues.* A multilogue consists of *unilogues*, each of which denotes the behaviour of a process. The unit of the behaviour, message reception, *etc.*, is called an *event.* A unilogue description begins with the expression "$< process\_name >=$," which is followed by an event sequence.

The message transfer events are indicated by expressions with a prefix of '+' or '-'. An event with prefix '+' indicates a message reception, while one with '-' indicates a message transmission. An identifier following the signs denotes the target process name. For example, the expression $+q(answer)$ in the unilogue of the process p says *process p receives message "answer" from process q.* When the target process name is omitted, for example $+(question)$, the expression indicates a message transfer from/to the private environment of the process. It is possible that no message is specified in the message transfer

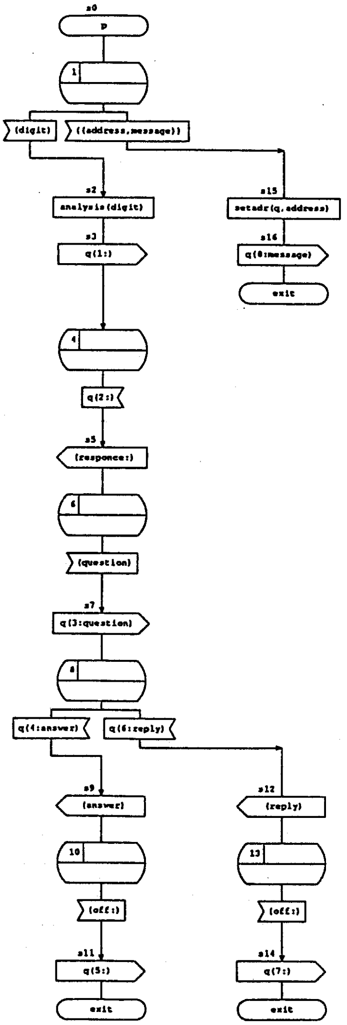


Figure 1 (a) SDL Chart

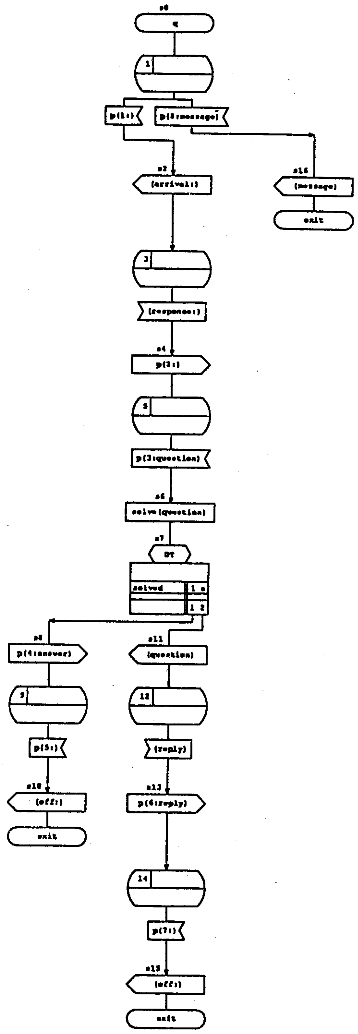
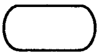
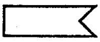
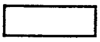
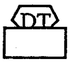
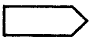


Figure 1 (b) SDL Chart

Table 1. Box types in SDL/GR

Box type	Meaning
	State. A state from where transitions are activated on the reception of a message
	Message Input.
	Task. A set of internal actions
	Logical decision. Originally, ◇
	Message Output.

```
     mesTransfer() {
   /* service 1 */
 p = +(digit) .analysis(digit) -q() +q() -(responce:)
     +(question) -q(question) +q(answer) -(answer) +(off:) -q() ;
 q = +p() -(arrival:) +(responce:) -p()
     +p(question) .solve(question) .if(solved)
     -p(answer) +p() -(off:) ;
|  /* service 2 */
 p = +(digit) .analysis(digit) -q() +q() -(responce:)
     +(question) -q(question) +q(reply)-(reply) +(off:) -q() ;
 q = +p() -(arrival:) +(responce:) -p()
     +p(question) .solve(question) .other -(question)
     +(reply) -p(reply) +p() -(off:) ;
|  /* service 3 */
 p = +((address,message)) .setadr(q,address) -q(message) ;
 q = +p(message) -(message) ;
}
%%
```

Figure 2 SAL program

to specify a message whose content is irrelevant. Message headers can also be omitted. The message reception $p(4 : answer)$ in Fig. 1 is described as $+p(answer)$ in Fig. 2. This is because SAL facilitates the description of each service without referring to other services. The SDE arranges the appropriate constant messages and message headers to combine service descriptions into SDL processes, preserving well-formedness of their branches. In SDL, branches in processes are restricted by decisions or by message receptions. For more detailed explanation, see [ICHI86].

A decision event is expressed by the form "$.if(< expression >)$". A task event must start with '.'. It is beyond the scope of this paper to present more detailed syntactical conditions for the above *expressions*, or the expression of *messages* to be transfered. More complex structures of the SAL programs and the event descriptions necessary for the subsequent discussion will be presented in the next subsection.

The concept of multilogues is a common one in the message sequence chart. The multilogues in Fig. 2 are described in the message sequence chart shown in Fig. 3. A vertical line in the chart represents the progress, from top to bottom, of a process behaviour. The meaning of the boxes are self-explanatory. The arrows between process lines denote message transfers between processes. One arrow represents both message transmission and the corresponding message reception.

2.2 Structure of SAL description

The expressive power of SAL is beyond that of the usual message sequence chart. The three features described below are unique to SAL. For the sake of simplicity, other facilities, such as the *declaration* of types, variables and so forth, are not presented in this paper.

(1) *Compound event:* The following is the syntax of a compound event:

$\{< multilogue >\}$, or
$\{< event\_sequence > \mid < event\_sequence > \mid ...\}$

The second case denotes the *choice* of alternative behaviours. The first case is a special case of the second. It is important that not only the event sequence but also the multilogue is allowed to occur in the compound event. Consider the unilogue below.

$$p = ...+r()\ \{p = +q(i)\ .a-q();\ q = -p(i)+p();\}-r(i)...;$$

This is equivalent to the following unilogue pair.

$$\begin{aligned} p &= ...+r()+q(i)\ .a-q()-r(i)...; \\ q &= -p(i)+p(); \end{aligned}$$

Generally, in case of unilogue $p = ...\{p = ...\}..;$, the unilogue header "$p =$" in the compound event can be omitted.

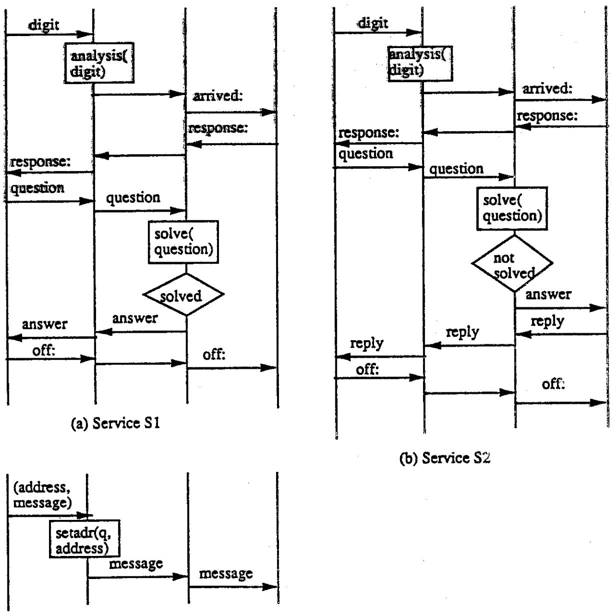
The next example contains an alternative choice.

$$\begin{aligned} p &= +q(i)\ \{.if(i >= 0)\ .a|.if(i < 0)\ .b\} - q(); \\ q &= -p(i) + p(); \end{aligned}$$

This expression is equivalent to the following:

$$\begin{aligned} p &= +q(i)\ .if(i >= 0)\ .a - q(); \\ q &= -p(i) + p(); \end{aligned}$$
$$|$$
$$\begin{aligned} p &= +q(i)\ .if(i < 0)\ .b - q(); \\ q &= -p(i) + p(); \end{aligned}$$

The exact meaning of the compound event will be clarified by the SAL expansion rules explained later.



(a) Service S1

(b) Service S2

(c) Service S3

Fig. 3 Service descriptions in the message sequence chart

(2) Multilogue call event: The multilogue set definition has the form

$$< name > (< parameter\_list >) < body >,$$

where $< body >$ is $\{< multilogue >\}$ or $\{< multilogue > \mid < multilogue > \mid ...\}$. One multilogue set can be referred to in another multilogue set. Consider the following pair of multilogue definitions:

$$\begin{aligned} m_1()\{ \quad p &= +r()\ .m_2() - r(i); \\ r &= -p() + p(i);\} \\ m_2()\{ \quad p &= +q(i)\ .a - q(); \\ q &= -p(i) + p();\} \end{aligned}$$

This pair is equivalent to the following single multilogue definition:

$$\begin{aligned} m_1()\{ \quad p = &+r()\ \{p = +q(i)\ .a - q();\ q = -p(i) \\ &+p();\} - r(i); \\ r = &-p() + p(i);\} \end{aligned}$$

Note that the form of the $< body >$ in the multilogue definition coincides with that of the compound event. An event like $.m_2()$, whose form is same as that of the task event, is called a *multilogue call event.*

(3) Loop event: This event has the form

$$.loop < compound\_event >,$$

which represents an iterative behaviour. Every unilogue in the compound event to be iterated must begin with either a message reception or a decision. This is because an iteration always induces a branch in the SDL chart.

A multilogue which holds to the following conditions is referred to as *simple.*

1. No unilogue includes a compound event which comprises alternatives.
2. No unilogue includes a multilogue call event.
3. Every unilogue $p = ...$ has only the events offered by p.

Any suit of multilogue set definitions can be translated into one in which every multilogue set definition comprises a suit of simple multilogues. Let us call this translation *expansion.* This expansion is performed by applying the following operation rules. These rules define the semantics of SAL.

1. **Macro expansion** The substitution of multilogue call events with compound events.
2. **Expansion of alternatives** The operation illustrated in the explanation of the compound event.
3. **Sequencing** Translation of a multilogue into a unilogue as described below. Let the following be a multilogue:

   $$\begin{aligned} p_1 &= E_1; \\ &\vdots \\ p_n &= E_n; \end{aligned}$$

   Note that not every event in sequence E_i is offered by p_i. The above multilogue is translated into the following unilogue:

   $$p_1 = E_1\{p_2 = E_2;\}...\{p_n = E_n;\};$$

4. **Filtering** Translation of a unilogue $p_1 = E$, obtained by using the above rules 1 through 3, into a simple multilogue

   $$\begin{aligned} p_1 &= E'_1; \\ &\vdots \\ p_m &= E'_m; \end{aligned}$$

   where E'_i is the subsequence of E, all of whose events are offered by p_i.

Example. Consider the next suit of multilogue sets.

$$\begin{aligned} m_1()\{ \quad p_1 = &\ \{+(a:) - p_2(a:) | + (b:) - p_2(b:)\} + p_3(); \\ p_2 = &+p_1(x)\ .m_2() - p_3(); \\ p_3 = &+p_2() - p_1();\} \\ m_2()\{ \quad p_2 = &\ .store(x)\ .loop\{+(c:y) - p_3(y);\ p_3 = \\ &+p_2(y)\ .store(y)\} + (a:);\} \end{aligned}$$

By applying rule 1, rule 2, rule 3 and rule 4 in this order, the following set of simple multilogues is obtained.

$$\begin{aligned} m_1()\{ \quad p_1 = &+(a:) - p_2(a:) + p_3(); \\ p_2 = &+p_1(y)\ .store(x)\ .loop\{+(c:y) - p_3(y)\} \\ &+(a:) - p_3(); \\ p_3 = &\ .loop\{+p_2(y)\ .store(y)\} + p_2() - p_1(); \\ &| \\ p_1 = &+(b:) - p_2(b:) + p_3(); \\ p_2 = &\ ...; \\ p_3 = &\ ...;\} \end{aligned}$$

Note the order of events in the unilogue for process p_3. □

3 GSAL

GSAL offers a graphic representation of SAL programs. GSAL involves some important extensions of the message sequence chart to support the SAL features shown in subsection 2.2. The elements of GSAL used to draw charts are listed below.

Process line Same as the vertical line of the message sequence chart.

Arrow Denotes a message transfer between processes. A pair of message expressions is attached to the arrow.

Event box A box on a process line represents an event other than the message transfer between processe.

Arrow: In a case where expression pair $a : x/y$ is attached to an arrow α from the line of process p to the line of process q, at the initial point of α, process p offers event $-q(a : x)$, and at the terminal point of α , q offers event $+p(y)$. If no expression pair is attached, the arrow denotes events $-q()$ and $+p()$. Concerning the validity of attached message expression pairs, see [ICHI86]. Note that the correspondence between send/receive events is represented by only arrow. This is an important feature of GSAL which SAL does not have. It is easy to preserve the consistency of interprocess communication in editing GSAL charts.

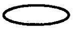
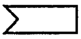
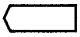
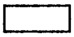
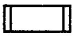
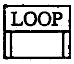
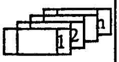
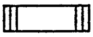
Event box: Types of event boxes are listed in Table 2. The *compound box*, the *selection box* and the *multilogue event box* are important elements which introduce a hierarchy into GSAL charts. Each of these represents subcharts. A selection box represents a compound event which comprises alternative behaviours. A selection box figured as n-repeated boxes denotes that there are n subcharts, each of which represents an alternative behaviour. The other type of compound event is represented by a compound box. The meaning of the multilogue event box is self-explanatory. A loop box is a selection box or a compound box with an accessary *loop*.

Example. Figure 4 illustrates the GSAL charts representing the following SAL program:

$$\begin{aligned} m_1()\{\ \ p =&\ \ +q(i)\ \{.if(i>=0)\ .a|.if(i<0)\ .b\} - q();\\ q =&\ \ -p(j)+q()\ .loop\{q=+(c:)-r(c:);\\ &\ \ r=+q(c:)\ .s;\}\{q=+(d:)-r(d:);\\ &\ \ r=+q(d:);\};\}\end{aligned}$$

Chart (a) is the top-level chart. Charts (b) and (c) are subcharts corresponding to the selection box of process p in the top–level chart. Charts (d) and (e) are subcharts coresponding to the loop box and the compound box of process q. □

Table 2 Local event boxes in GSAL

Event box	name
	Process header
	Message reception from the environment
	Message output to the environment
	Task
	Compound box
	Loop box
	Selection Box
	Multilogue event Box

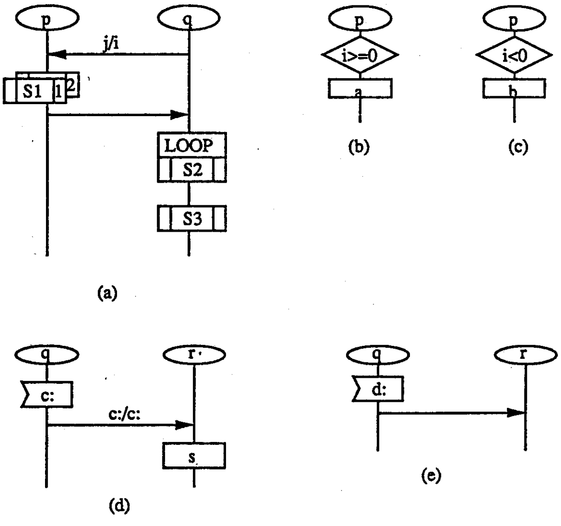


Figure 4 GSAL charts

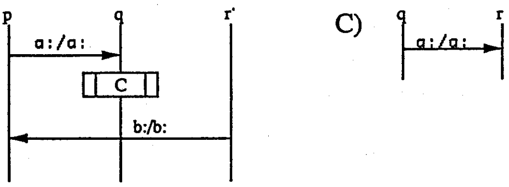


Figure 5 Ambiguous GSAL charts

Restriction on GSAL charts: Consider the following example:

$$\begin{aligned} p &= -q(a:) - r(b:);\ /*u_1*/ \\ q &= +q(a:)\{-r(a:);\ r = +q(a:);\};\ /*u_2*/ \\ r &= +r(b:);\ /*u_3*/ \end{aligned}$$

We know that event $+r(a:)$ is offered by process r before $+r(b:)$. This order, however, is not well represented by the GSAL chart in Fig. 5 which is a direct interpretation of the above multilogue. This is because both unilogues u_2 and u_3 include events offered by process r. From the top-level chart only, we may guess that the event $+r(b:)$ is to be offered first by r.

For this reason, the following restriction is placed on GSAL charts. Let $P(u)$ be the set of processes which offers some events in unilogue u. For the above multilogue, $P(u_1) = \{p\}$, $P(u_2) = \{q,r\}$ and $P(u_3) = \{r\}$. The restriction is that for a multilogue consisting of unilogues $u_1, u_2, \ldots, u_n$,

$$P(u_i) \cap P(u_j) = \emptyset$$

for any pair of distinct unilogues. Under this restriction, the order of event offers is completely represented by the GSAL chart.

4 GSAL Generation

Text forms and graphic form sof languages each have their own advantages. For example, a text form is well suited for writing massive programs. The graphic form is superior to the text form in readability. It is also good for program modification. In cases where a language has both forms, it is efficient that translation between the two forms is possible.

SAL generation from GSAL is straightforward. However, two steps must be performed for the converse translation. One is to determine the correspondence between send/receive events in order to draw the necessary arrows. This steps comprises a sort of protocol verification, and is therefore called *protocol check.* The second step is to modify the SAL program so that its GSAL chart obeys the restriction described in the previous section. This step is called *reforming.*

Protocol check: This step is performed by using the simple multilogue set derived from a given SAL program. The protocol check is easy for simple multilogues, as shown by the example below. Consider the following multilogue:

$$\begin{aligned} p_1 &= +(a:) - p_2(a:) + p_3(b:); \\ p_2 &= +p_1(x)\ .store(x)\ .loop\{+(c:y) - p_3(y)\} \\ &\quad +(a:) - p_3(b:); \\ p_3 &= .loop\{+p_2(z)\ .store(z)\} + p_2(b:) - p_1(b:); \end{aligned}$$

By simulating the multilogue, the following correspondences are obtained:

$$\begin{array}{llllll} -p_2(a:) & \longrightarrow & +p_1(x), & -p_3(y) & \longrightarrow & +p_2(z) \\ -p_3(b:) & \longrightarrow & +p_2(b:), & -p_1(b:) & \longrightarrow & +p_3(b:) \end{array}$$

In general, it is necessary for every interprocess communication event to be related to another event, as in the above multilogue. If this condition is not met, it is concluded that the SAL program is not valid. (See [ITOH86] for more detailed techniques of the protocol checking.) It is also easy to obtain send/receive correspondences for the given SAL program from the results of its equivalent simple multilogue set. The only difficulty is the case where the members of an event pair inducing an arrow belong to different compound events or multilogues. In those cases, it is necessary to identify the boxes for the events to be involved in order to induce an arrow between a process line and a box or between boxes. For this identification, box names are attatched to every event when expanding the given multilogue into the simple multilogue set.

Reforming: For this step, the process list $P(u)$ is necessary to calculate each unilogue of the given multilogue. This calculation is defined as follows: Let unilogue u be $p = E$, where E is an event sequence, and, let E be "$e\ E'$," where e is an event. Then,

$$P(u) = \{p\} \cup P(E),\ P(E) = P(e) \cup P(E').$$

When e is a compound event or a multilogue call event, $P(e)$ is defined as shown below. Otherwise, $P(e) = \emptyset$.

$$P(\{E_1|E_2|\cdots|E_n\}) = P(E_1) \cup P(E_2) \cup \cdots P(E_n),$$

$$P(\{u_1;u_2;\cdots u_m;\}) = P(u_1) \cup P(u_2) \cup \cdots p(u_n),$$

where E_i, $i = 1,\ldots,n$, and u_j, $j = 1,\ldots,m$ are event sequences and unilogues, respectively.

Let $u_1;u_2;\cdots,u_k;$ be a multilogue. Reforming is performed with the following algorithm:

```
calculate$P(u_k)$;
$i := k - 1$;
while($i > 0$) do
   begin
      calculate $P(u_i)$;
      $m := i + 1$;
      while($m \leq k$) do
         begin
            if($P(u_i) \cap P(u_m) \neq \emptyset$)do
               begin
                  $u_i := u_i\{u_m;\}$;
                  $P(u_i) := P(u_i) \cup P(u_m)$;
                  remove $u_m$; }
               end
            $m := m + 1$;
         end
      $i := i - 1$;
   end
```

It is obvious that the new multilogue obtained via the above reforming algorithm is equivalent to the original one and obeys the restriction of the GSAL chart.

Example. Let m be the following multilogue:

$$\begin{aligned} p_1 &= +(a:) - p_2(a:)\ .loop\{+(b:x) - p_4(b:x); \\ &\quad p_4 = +p_1(b:y)\ .store(y);\}\{+(c:) - p_4(); \\ &\quad p_4 = +p_1();\} + p_3();\ /*u_1*/ \\ p_2 &= +p_1(a:)\ .m_1;\ /*u_2*/ \\ p_3 &= .t - p_1();\ /*u_3*/ \\ p_4 &= .s;\ /*u_4*/ \end{aligned}$$

The multilogue m_1 is as follows:

$$\begin{aligned} p_2 &= -p_3(i) + p_3(); \\ p_3 &= +p_2(j)\ .k(j) - p_2(); \end{aligned}$$

The process name list $P(u_i)$ are calculated as follows:

$$P(u_1) = \{p_1, p_4\},\ P(u_2) = \{p_2, p_3\},$$

$$P(u_3) = \{p_3\},\ P(u_4) = \{p_4\}$$

Using those lists, multilogue m is reformed into the following:

$$u_1\{u_4;\};\ u_2\{u_3;\};$$

□

5 Conclusion

The fundamental features of GSAL and its conversion from text-form SAL have been presented. GSAL is an extended form of the message sequence chart which has been widely used in telecommunications service design. The extention mainly introduces a hierarchical structure and an iteration structure.

GSAL is superior to SAL in readability, but also, arrows in GSAL charts explicitly express the correspondence between send/receive events offered by concurrent processes. Therefore, GSAL generation from SAL performs a sort of protocol checking between processes, which is a major considerations in telecommunications services.

Acknowledgement The authors thank H. Ishimoto for his cooperation in the GSAL system implementation. They would also like to express their gratitude to colleagues at British Telecom, L.A. Jackson, M.T. Norris and R. Tinker. This project was initiated through collaboration on the SDE.

References

[ASN188] *ASN.1* ISO 8824, 8825.

[BOCH80] G. v.Bochmann, *A general transition model for protocol and communication services,* IEEE Trans. on Commun., Vol. COM-28, No. 4, pp.643-650, Apr. 1980.

[BRIN86] Ed Brinksma, G. Scollo and C. Steenbergen, *Lotos specifications, their implementations and their test,* Protocol Specification, Testing, and Verification VI (B. Sarikaya and G. v.Bochmann, eds.), North-Holland, pp.349-360, 1986.

[ESTE88] *Estelle,* ISO 9074.

[ICHI86] H. Ichikawa, M. Itoh and M. Shibasaki, *Protocol-oriented service specifications and their transformation into CCITT specification and description language,* Trans. of IECE of Japan, Vol. E69, No.4, pp.524-535, Apr. 1986.

[ITOH86] M. Itoh and H. Ichikawa, *Hierarchical verification of communications programs based on concurrent processes,* Trans. IECE Japan, Vol. J69-B, No. 5, pp.449-459, May 1986.

[LOTO88] *LOTOS,* ISO 8807.

[SCOL86] G. Scollo and M. v.Sinderen, *On the architectural design of the session standard in Lotos,* Protocol specification, Testing, and Verification VI (B. Sarikaya and G. v.Bochmann, eds), North-Holland, pp.3-14, 1986.

[SDL84] CCITT. *Specification and Description Language,* Red Book, Recommendation Z.100 - Z.104, Vol. VI, Fasc. 10-11, 1984.

[VISS85] C.A. Vissers and L. Logrippo, *The importance of the service concept in the design of data communication protocols,* Protocol Specification, Testing and Verification V (M. Diaz, ed), North-Holland, pp.3-17, 1985.