Deriving Use Cases for Distributed Systems from Knowledge Requirements[1]

Xiao Jun Chen
School of Computer Science
University of Windsor
Windsor ON Canada
xjchen@cs.uwindsor.ca

Luigi Logrippo
School of Information Technology and Engineering
University of Ottawa
Ottawa, ON Canada
luigi@site.uottawa.ca

Abstract

Knowledge requirements for distributed systems express desired states of affairs concerning initial and final knowledge of the components of a system. It is shown how *use cases* (i.e. sets of scenarios) can be obtained from knowledge requirements. The technique involves the use of *event structures* that specify use cases capable of achieving the requirements, and logical postconditions for events. The technique allows design refinement, both in the form of architectural refinement and in the form of event refinement. The correctness of refinements can be checked by checking logical implications. As an example, use cases and scenarios for a simple mobile telephony protocol, based on GPRS (the data extension of GSM) are derived, on the basis of the corresponding knowledge requirements. Scenarios are given in the form of Message Sequence Charts.

Keywords: use cases, scenarios, logic specifications, knowledge requirements, event structures, distributed system design, protocol design.

1. Introduction

Use cases have been defined [Jac92, BuC95] as structured prose descriptions of interaction *scenarios* between a system to be designed and users of the system. Thus use cases are sets of scenarios. They have been recognized as helpful in systems design, test case generation, etc. The formulation of use cases is often taken as the initial step in the software design process, in the sense that use cases specify system functionalities and system requirements [ALBG99].

In this paper, we propose a somewhat more fundamental view. We start from the idea that *the function of a distributed system is to distribute knowledge*. System functionalities start with a situation where some agents know something, and aim to achieve a situation where other agents know the same, or a related, thing. For example, when Joe knows that he has won a lottery, he may use the telephone system to share this information with Mary. This common knowledge is the final postcondition that Joe intends to achieve by using the telephone system. The use of this system, as well as its internal functioning, involves a large number of intermediate knowledge requirements. By the way telephone systems are generally conceived (which we can take as a preexisting *architectural constraint*), the user's requirement can be achieved by going through an intermediate step where the user and the system share knowledge of Mary's telephone

1. Appeared in Annals of Telecommunications 54, 11-12, 1999, 2-13

number.This becomes an intermediate requirement or postcondition, that must be achieved in order for the overall requirement to be achieved. This intermediate requirement must in its own turn be decomposed, normally by first reaching a situation where the system knows that the user is about to send the number, and the user knows that the system is ready to accept it.

In other words, by lifting the handset, the system will know that Joe wants to talk; by hearing the dial tone, Joe knows that he can dial; by dialing, the system will know the address of Mary's phone, etc., until the final requirement is reached (or not if Mary is absent or busy, two other scenarios). The well-known use cases of basic telephone connection can be constructed in terms of knowledge requirements, architectural constraints, and design refinements necessary to meet the architectural constraints at increasing levels of refinement, while keeping the preexisting knowledge requirements invariant.

According to this philosophy, a first step towards designing systems functionalities and the corresponding use cases is determining who must know what and in what order. The physical and engineering constraints of real-life systems will require refinements in the form of intermediate agents and intermediate requirements. We show that use cases and scenarios can be derived from this information by using a refinement process.

We consider the functionalities to be implemented one by one: for each functionality, we have a set of *knowledge requirements* it is intended to achieve. We derive use cases and scenarios that satisfy the knowledge requirements under consideration, as well as existing facts such as system topology (which are the *architectural constraint*), and other knowledge about the system (which we call *general knowledge*).

To specify knowledge requirements, we use English statements. Ideally, these should be formalizable following the theory of epistemic logic [Hin62] towards distributed systems [Hal87,HF89], but we do not provide this formalization in the paper. We exploit *event structures* [Wi80, Wi89, BoCa89, La91] as a way of representing use cases and we show how to derive them from knowledge requirements step by step. Event structures have been invented on the basis of mathematical intuitions rather than design considerations. However there are intuitions behind event structures that are basically simple and graphical in nature. In this paper, we slightly extend this notation according to the nature of the use cases.For people leery of anthropomorphic language, *knowledge* can easily be translated to *availability of information.*

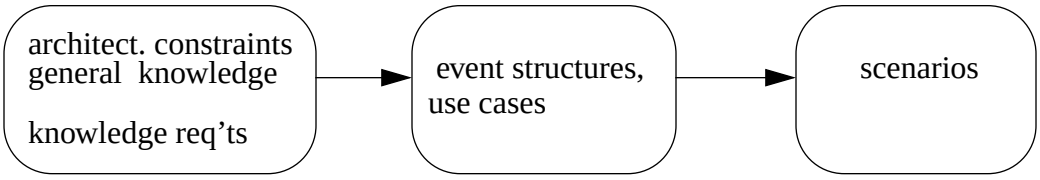


Fig. 1. Obtaining use cases and scenarios

The method we propose is illustrated in Figure 1: (i) we assume the existence of a number of facts, called *general knowledge* and *architectural constraints*; (ii) for each functionality, we have *knowledge requirements* corresponding to it; (iii) from these, we construct event structures to represent functionalities; as we have mentioned, we take each event structure as a use case corresponding to a functionality; (iv) from the event structure, a set of scenarios can be derived. The most important step is (iii): we show the construction of the event structures in a stepwise refinement method. In doing so, we introduce a set of postconditions associated to each event in

the event structure. Using preconditions (assumptions) and postconditions (consequences) to prove the correctness of refinements with respect to a specification has been extensively studied for sequential programs, while to deal with concurrent systems, [AL93] have discussed proving the properties (consequences) of a composite system from the properties (assumptions) of the relative behaviors of its parallel components. Here we show that the postconditions that we introduce facilitate the refinement of event structures from one step to another.

As already hinted above, it should be noted that our work is not limited to the derivation of use cases and scenarios, but also addresses the more general problem of system design. However in order to be complete in this respect, the method should be capable of expressing iteration or recursion, which require further study.

In Sections 3 and 4, a sizeable example of use of our method is provided. It is based on the forthcoming standard GPRS, the data extension of GSM.

Section 2. Extended stable event structures

Event structures [Wi80, Wi89] have achieved recognition as truly concurrent models for process theory [BoCa89, La91]. The intuitions behind event structures are basically simple and graphical in nature, and we show here that they provide a suitable way for formal requirement definition in distributed system designs.

Various kinds of event structures have been studied in the literature, such as prime event structure [Wi80, Wi89], stable event structure [Wi89], flow event structure [BoCa89], bundle event structure [La91]. In this paper, we adopt some basic notions from stable event structures, and extend them for the definition of functionalities, use cases and scenarios.

A *stable event structure* consists of events and relations between events. The events model the occurrence of actions, the fact that “something happens”. The events are labelled: each event has a label denoting an action. Different events are represented by different circles, but they may have the same label. We represent events by small circles ○ . In particular, we use in our extended version, a circle ⊙ with a dot inside to denote an *initial event*, and a black circle ● to denote a *final event*. This information will be required when we obtain scenarios, as will be presented later.

There are mainly three kinds of relations between events: the enabling relation, the conflict relation, and the independence relation (in what follows ○ denotes an arbitrary circle, which could also be ⊙ or ●).

(1) *The enabling relation*.

An enabling relation is an antisymmetric relation between a set of events *F* and another event *e*. The intuition behind is that event *e* is enabled once all events in *F* have happened. We depict this relation by drawing an arrow from each element of *F* to *e* and connecting all the arrows by small lines. For example, in Figure 2(1), *e* is enabled by *a* and *b*. In other words, in each system run *r*, if *e* appears, then both *a* and *b* should appear in *r* before *e*. Note that there may be more than one set of events to enable the same event. In Figure 2(2), *e* can be enabled either by both *a*

and *b* or enabled by *c* only.

Figure 2. Examples for the enabling relation

(2) *Conflict relation.*

A conflict relation is a symmetric binary relation between events. We use a ○- - - -○b to denote the conflict relation between event *a* and event *b*. The intuitive meaning is that *a* and *b* will never happen together in any single system run.

(3) *Independence relation*.

An independent relation is a symmetric binary relation between events. We use a ○ ○ b (the absence of enabling and conflict relation) to denote that event *a* and event *b* are independent. That is, if both *a* and *b* are enabled, they can occur in any order or simultaneously, so they neither need to happen in parallel nor to occur one before the other.

Stable event structures must satisfy consistency and stability properties as defined in [La91].

In our refinement process of deriving an event structure from knowledge requirements, an event can be either a *primary* one or a *composite* one. A primary event is either an internal event of an agent, or a message passing from one agent to another. A composite event, on the contrary, cannot be implemented directly. It must be *decomposed* according to the requirements, until all the events are primary ones. We associate each event with a set of postconditions that are true after the event. We use {*p1*,*p2*,...,*pn*} to denote the conjunction of the postconditions *p1*, *p2*, ..., *pn*.

Postconditions are introduced to facilitate the refinement process. Let *a* be a composite event. Before *a* is refined, some parts other than *a*, including events that depend on *a*, may need to be refined. The postconditions of *a* can then be used as the preconditions to refine the events that depend on *a*. Later on, when *a* is refined, the postconditions of *a* can be used as the scope of *a*'s decomposition. The refinement of an event structure can be the decomposition of events or the refinement of postconditions, or both of them. The fact that a postcondition *p'* refines a postcondition *p* means that *p'* implies *p*. This refinement is considered to be correct because it does not change the possibility of enabling any events that depend on *p*.

Section 3. An example: the GPRS system

In this section, we provide an example of how to derive extended stable event structures using postconditions. For simplicity, such structures with postconditions will also be called event structures. We take one of the functionalities of GPRS, namely the *routing update* procedure. We discuss the knowledge requirements for it, and we exemplify the derivation of an event structure for it.

This section can be described as an exercise in *reverse engineering*, since we are taking an existing system and we are trying to justify *a posteriori* its design on the basis of our method.

Note that our example is based on ETSI documents on GPRS that are still being modified. People active in the development of this standard may recognize our description as outdated, however this is not important for the purpose of this paper.

Section 3.1 GPRS and main architectural constraints

GPRS (the General Packet Radio Service) is a packet switched service that is currently being developed by European Telecommunications Standardization Institute (ETSI)[ETSI98]. It is being implemented on top of the existing Global System for Mobile communication (GSM) [MP92]. It provides data transmission services for mobile subscribers and allows users to be involved in either Point-to-Point or Point-to-Multipoint transmissions.

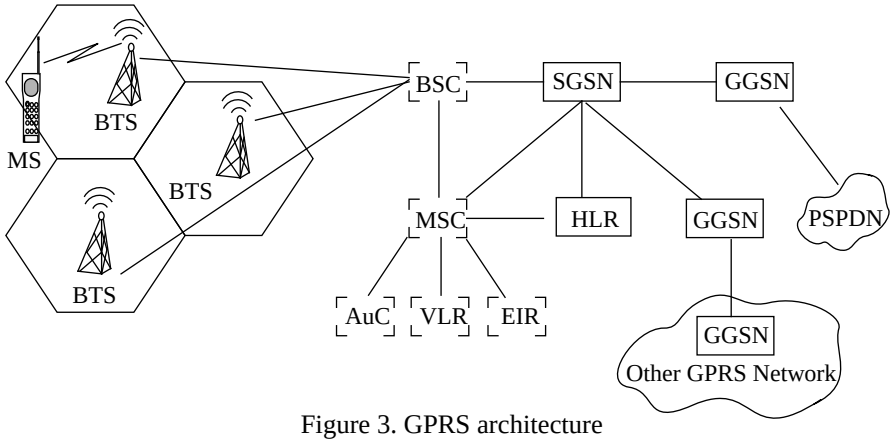


Figure 3. GPRS architecture

Figure 3 shows a simplified GPRS architecture. Here

1. A Mobile Station (MS) is a terminal equipment that the subscriber uses to access the services from the network; MSs find themselves in *cells* (represented by hexagons) and *location areas* (concepts developed in Section 3.2).
2. The Home Location Register (HLR) contains the subscription information such as service profiles, identifications of MSs and the MSC areas in which the MSs are currently located. Each subscriber is registered in only one HLR which is the one that performs the charging and billing functions;
3. The Serving GPRS Support Node (SGSN) is the node that is serving the MS. It contains the Routing Context for the MSs attached to it. The Routing Context holds the information of the GGSNs (see below) that the GPRS subscribers will be using;
4. The Gateway GPRS Support Node (GGSN) is the node which is accessed by the packet data network due to the evaluation of the Packet Data Protocol (PDP) address. It contains a record for each attached GPRS user. Such records are used to communicate with the SGSNs who are serving the MSs. PSPDN denotes an external packet switched public data network.

The figure shows several other protocol entities (agents) in the architecture denoted by dashed boxes, i.e. the Base Station Transceiver (BTS), the Base Station Controller (BSC), the Mobile Switching Center (MSC), the Authentication Center (AuC), the Visitors Location Register (VLR), and the Equipment Identity Register (EIR). We do not introduce them since we do not discuss them in the paper. Our example is simplified and it takes into account only the logical links between MS, SGSN, GGSN, and HLR. Also, details related to authentication and security checks are not discussed.

From the above architecture we know, for example, that message passing may happen between an MS and an SGSN (provided that there is a link established between them, or in other words, they know the address of each other), while a HLR cannot talk to a GGSN. So we have the architectural constraints illustrated as follows (the arrows represent the connections, and the

loop on SGSN represents the fact that several interconnected SGSNs can be instantiated, see Fig. 6):

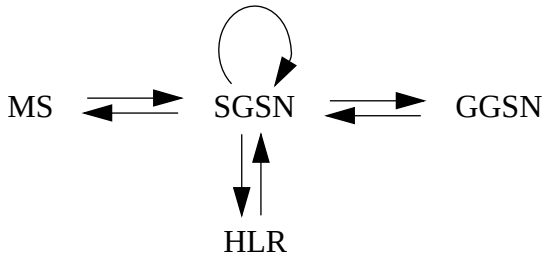


Figure 4. Interconnections among the entities

There is also other information related to the architecture. We review what is needed for our example.

When a user subscribes to the GPRS system, the information related to its subscription, the Quality of Service (QoS) for example, is stored in the HLR (i.e. it is known by it). This information is called SubscriberData, and whenever the subscriber is connected to an SGSN or switched to a new SGSN, such information is also stored in this (new) SGSN.

The Routing Context contains the necessary information to allow the transactions of sending and receiving messages between an MS and an external network. It is kept in the SGSN and updated every time the MS moves to a new location.

Among all the information that we can obtain from the GPRS system, we use the following *general knowledge*:

(A1) For any network entities *A* and *B*, *A* can talk to *B* only if *A* knows the address of *B*;
(A2) SGSN can only talk to MSs that are in its own location area;
(A3) All SGSNs know the address of the HLR;
(A4) The HLR knows the SubscriberData for each MS;
(A5) The SGSN in charge of an MS knows the RoutingContext of this MS;
(A6) Each MS knows the address of its SGSN.

Section 3.2 Knowledge requirements and design refinement

Mobility of subscribers makes it crucial for the network to be aware of the location of the user when it indicates its intention to use the services provided by the network. A mechanism has to be identified in order to keep track of the location of a Mobile Station in the network. This mechanism is called "*Routing Update*" and it is the subject of our example: we consider the knowledge requirements for such Routing Update procedure.

A Mobile Station moves around. At a certain point, it knows that it has changed *cell*. A *Location Area* contains a collection of geographically connected cells. Each Location Area has only one SGSN serving it. When the Mobile Station moves from one cell to another, its Location Area may change, implying a change of the SGSN to serve it. Thus, when the Mobile Station knows

that it has changed its cell, it starts an update procedure, resulting in the fact that eventually the New SGSN will know its new cell location.

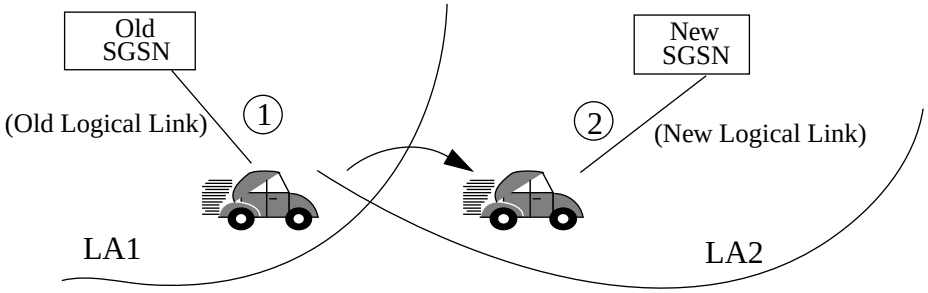


Figure 5. Routing update with change of SGSN

Assume then that a Mobile Station aMS has changed Location Area. Let aOldSGSN be the SGSN that was serving the aMS before it moved, and aNewSGSN be the SGSN that is in charge of serving the aMS after it moved. We consider the simple case where the aMS is connected to only one GGSN and we call it aGGSN. The related HLR is called aHLR. Among these entities, the interconnections are shown in Fig. 6.

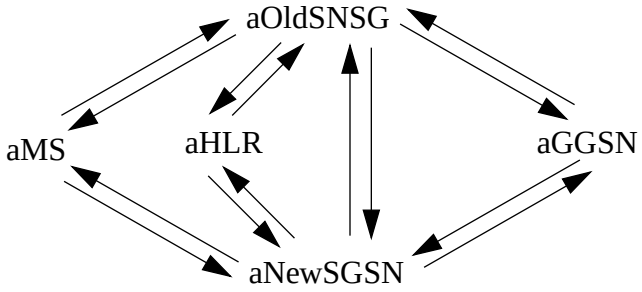


Figure 6. Interconnections among the entities of the routing update example

Now we show in four steps the derivation of a possible event structure that meets the *architectural constraints* in Figure 6, the general knowledge *(A1)--(A6)*, and the *knowledge requirements* (given below) for the Routing Update procedure.

Step (i).

When aMS moves to another Location Area, according to general knowledge *(A2)* and architectural constraint (Figure 6), aNewSGSN and only aNewSGSN can be informed of this change. So we have requirement

(C1) When aMS moves to the Location Area of aNewSGSN, it should inform aNewSGSN of its change of Location Area;

Receiving the information that aMS has moved to the new Location Area, aNewSGSN knows that a routing update is required, and it can broadcast the update request to other protocol entities.

Both aGGSN and aNewSGSN have the right to reject the routing update of the aMS. They may reject the request if, due to the regional, national or international restrictions, aMS is not allowed to roam into the Location Area or if the subscription check fails. If aNewSGSN rejects the request, since aNewSGSN cannot serve aMS while it is the only possible entity to talk to

aMS, then aMS should receive a negative notification (it has no logical link to aNewSGSN). Otherwise, aMS should receive a positive notification (it has logical link to aNewSGSN), and it should also be informed whether it has connection to aGGSN (whether it can send and receive data by way of aGGSN). Precisely,

(C2) For each update request, aNewSGSN could either accept the update (Upd-Acc) or reject the update request (Upd-Rej);
(C3) If aNewSGSN rejects the request, then aMS should be notified that it has no logical link to aNewSGSN (Upd-Notify-);
(C4) If both aNewSGSN and aGGSN accept the request, then after the update is done (Update1), aMS should be notified that it has logical link to aNewSGSN and that it has connection to aGGSN (Upd-Notify+1);
(C5) If aNewSGSN accepts the request while aGGSN rejects it, then after the update is done (Update0), aMS should be notified that it has logical link to aNewSGSN but (currently) it has no connection to aGGSN (Upd-Notify+0).

Note that according to the architectural constraint, and the general knowledge, aNewSGSN is the only entity who could notify aMS of the update result. So we can rewrite *(C4)* and *(C5)* as

(C4') If both aNewSGSN and aGGSN accept the request, then after the update is done (Update1), aNewSGSN should tell aMS that it has logical link to aNewSGSN and that it has connection to aGGSN (Upd-Notify+1);
(C5') If aNewSGSN accepts the request while aGGSN rejects it, then after the update is done (Update0), aNewSGSN should tell aMS that it has logical link to aNewSGSN but (currently) it has no connection to aGGSN (Upd-Notify+0).

In order to tell aMS these corresponding results, aNewSGSN itself should know them (postconditions *nkud, nkce, nkcn*) after the update. Thus, we have the event structure shown in Figure 7. Note that for each event, we give in the figure a name of the event together with names of each of its postconditions, whose meanings are given in the corresponding Table 1. The dashed ellipse in the figure, as well as the events with "*" in the table, identify the composite events that are to be refined in successive steps.

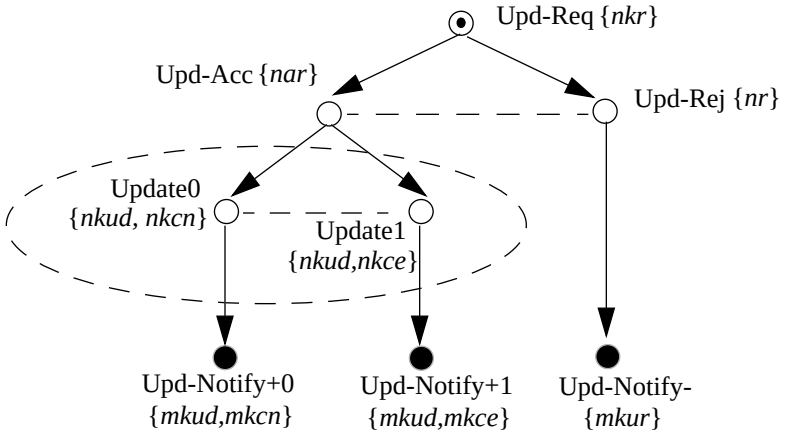


Figure 7. Event structure for routing update: step (i)

Table 1: Events and postconditions in Fig. 7

event name	event	post cond name	postconditions
Upd-Req	aMS tells aNewSGSN it has changed Location Area	*nkr*	aNewSGSN knows the update request for aMS
Upd-Acc	aNewSGSN accepts the update request	*nar*	aNewSGSN accepted the update request
Upd-Rej	aNewSGSN rejects the update request	*nr*	aNewSGSN rejected the update request
Update0	*GPRS does the routing update with no connection to aGGSN established, and notifies aNewSGSN	*nkud* *nkcn*	aNewSGSN knows aMS has logical link to aNewSGSN; aNewSGSN knows aMS has no connection to aGGSN
Update1	*GPRS does the routing update with connection to aGGSN established, and notifies aNewSGSN	*nkud* *nkce*	aNewSGSN knows aMS has logical link to aNewSGSN; aNewSGSN knows aMS has connection to aGGSN
Upd-Notify+0	aNewSGSN tells aMS that it has logical link to aNewSGSN but it has no connection to aGGSN	*mkud* *mkcn*	aMS knows it has logical link to aNewSGSN; aMS knows it has no connection to aGGSN
Upd-Notify+1	aNewSGSN tells aMS that it has logical link to aNewSGSN and it has connection to aGGSN	*mkud* *mkce*	aMS knows it has logical link to aNewSGSN; aMS knows it has connection to aGGSN
Upd-Notify-	aNewSGSN tells aMS that it has no logical link to aNewSGSN	*mkur*	aMS knows it has no logical link to aNewSGSN

Step (ii)

Now we go into the details about the update. To refine events Update0 and Update1 means to realize their postconditions *nkud, nkce* and *nkcn*. We say that

(C6) (*ud*) aMS has logical link to aNewSGSN
means that
(*nas*) aNewSGSN added the SubscriberData for aMS;
(*hu*) aHLR updated its Location of aMS.

(C7) (*ce*) aMS has the connection to aGGSN
means that
(*ur*) aNewSGSN added its RoutingContext of aGGSN for aMS;
(*gu*) aGGSN updated its record for aMS and aNewSGSN.

(C8) (*cn*) aMS has no connection to aNewSGSN
means that
(*gd*) aGGSN deleted its record for aMS and aOldSGSN.

We use $\frac{p1, p2, ..., pn}{p}$ to denote that from the conjunction of conditions *p1, p2, ..., pn*, we

can deduce condition *p*. So requirements *(C6)--(C8)* are expressed as (Fig. 8)

nas: aNewSGSN added SubscriberData for aMS,
hu: aHLR updated the Location of aMS,
---
ud: aMS has logical link to aNewSGSN

ur: aNewSGSN added its RoutingContext of aGGSN for aMS,
gu: aGGSN updated its record for aMS and aNewSGSN
---
ce: aMS has connection to aGGSN

gd: aGGSN deleted its record for aMS and aOldSGSN
---
cn: aMS has no connection to aGGSN

Figure 8. Requirements *(C6)--(C8)*

Furthermore, let *A* be a protocol entity, we assume that

$$\frac{p1, p2, ..., pn}{p} \quad \text{implies} \quad \frac{A \text{ knows } p1, A \text{ knows } p2, ..., A \text{ knows } pn}{A \text{ knows } p}$$

This implication can be easily derived from Epistemic Logic [Hin62]. Here we just use it by intuition. So from Figure 8 we obtain

nas: aNewSGSN added SubscriberData for aMS,
nkhu: aNewSGSN knows aHLR updated the Location of aMS,
---
nkud: aNewSGSN knows aMS has logical link to aNewSGSN

nur: aNewSGSN added its RoutingContext of aGGSN for aMS,
nkgu: aNewSGSN knows aGGSN updated its record for aMS and aNewSGSN
---
nkce: aNewSGSN knows aMS has connection to aGGSN

nkgd: aNewSGSN knows aGGSN deleted its record for aMS and aOldSGSN
---
nkcn: aNewSGSN knows aMS has no connection to aGGSN

Figure 9. Refinement of the postconditions

Note that above we have used *aNewSGSN added SubscriberData for aMS* (*nas*) to substitute *aNewSGSN knows aNewSGSN added SubscriberData for aMS*, since the former implies the latter. The same substitution is used for *nur*. It is interesting here to note that we distinguish between the situation where an entity knows some data and the situation where the entity knows that it has added the data to its database.

Thus, we obtain the refined event structure (cf. Fig. 10) on this step, where the postconditions (*nkud, nkce, nkcn*) of Update1 and Update0 are replaced by the conjunction of the conditions that imply them (cf. Figure 9). For example, *nkud* is replaced by the conjunction of *nas* and *nkhu*. The ellipse in the event structure singles out the events that are refined from the

previous step. Table 2 shows the meanings of the refined postconditions.

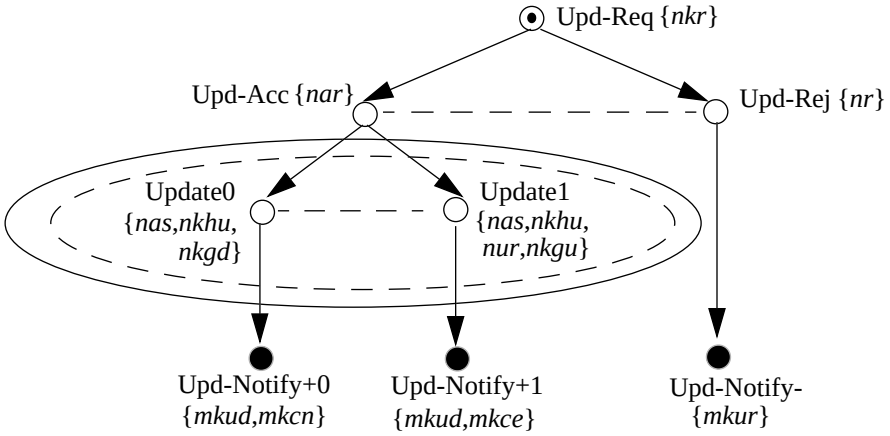


Figure 10. Event structure for routing update: step (ii)

Table 2: Events and postconditions in Fig. 10

event name	event	post cond name	postconditions
Update0	*GPRS does the routing update with no connection to aGGSN established, and notifies aNewSGSN	*nas* *nkhu* *nkgd*	aNewSGSN added SubscriberData for aMS; aNewSGSN knows aHLR updated the Location of aMS; aNewSGSN knows aGGSN deleted its record for aMS and aOldSGSN
Update1	*GPRS does the routing update with connection to aGGSN established, and notifies aNewSGSN	*nas* *nkhu* *nur* *nkgu*	same as above; same as above; aNewSGSN added the RoutingContext of aGGSN for aMS; aNewSGSN knows aGGSN updated its record for aMS and aNewSGSN

Step (iii)

Now we consider the newly obtained postconditions (*nas, nkhu, nkgd, nur, nkgu*) in the previous step.

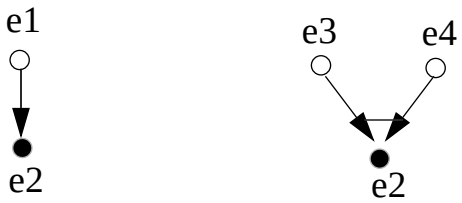
In our context, suppose we have relation *e1 enables e2.* As we know, this means that *postcond(e1)* enables *e2*. Here we use *postcond(e)* to denote the set of postconditions of *e*. During the refinement, we can

1. refine *e1* into *e3* and *e4*, to obtain the postconditions of *e1* by different events. I.e.

$$postcond(e3) \cup postcond(e4) \supseteq postcond(e1) \qquad (C)$$

2. refine relation *e1 enables e2* into *e3 and e4 enable e2*. As we know, *e3 and e4 enable e2* means that *postcond(e3)* $\cup$ *postcond(e4)* enables *e2*. Due to *(C)* above, this implies that *e1*

enables e2.



Now consider relation

Update0 {nas,nkhu,nkgd} enables Upd-Notify+0 {mkud,mkcn}

in the previous step. We refine *Update0* into *Upd-O0 {nkgd}, Upd-H{nkhu}* and *Upd-NS {nas}*. Obviously,

postcond(Upd-O0) ∪ *postcond(Upd-H)* ∪ *postcond(Upd-NS) = postcond(Update0)*

and we refine relation *Update0 enables Upd-Notify+0* into *Upd-O0, Upd-H and Upd-NS enable Upd-Notify+0*:

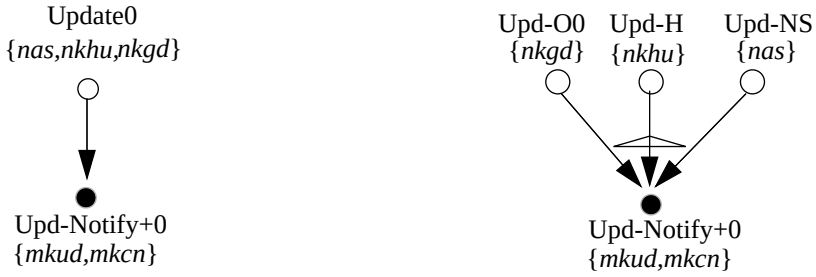


Figure 11. Obtaining a set of postconditions from several sets of postconditions (a)

Similarly, we refine event *Update1 {nas,nkhu,nur,nkgu}* in the previous step into events *Upd-O1 {nkgu}, Upd-H {nkhu}, NUG {nur}* and *Upd-NS {nas}*. Note that we have requirement

(C9) aNewSGSN can add its RoutingContext for aMS (NUG) only after it knows aGGSN has updated its record for aMS and aNewSGSN (*nkgu*).

So event NUG depends on event Upd-O1. Thus, we have

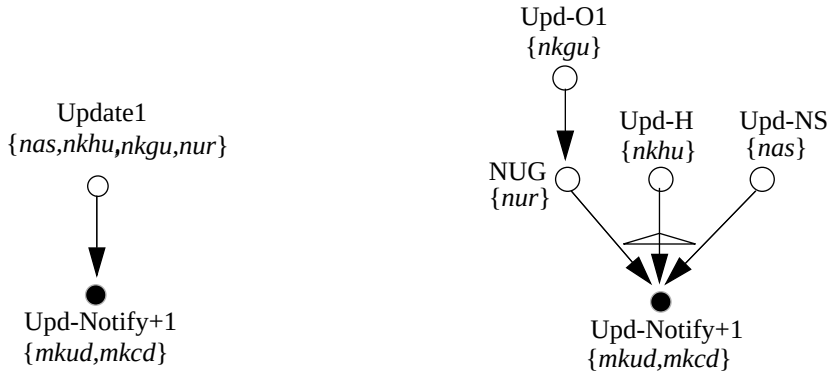


Figure 12. Obtaining a set of postconditions from several sets of postconditions (b)

Now considering Figure 11 and Figure 12, we obtain the structure on this step as shown in Figure 13 and Table 3 (with only explanations of the refined events). Note that events *Upd-H*

and *Upd-NS* in Figure 11 and Figure 12 are the same.

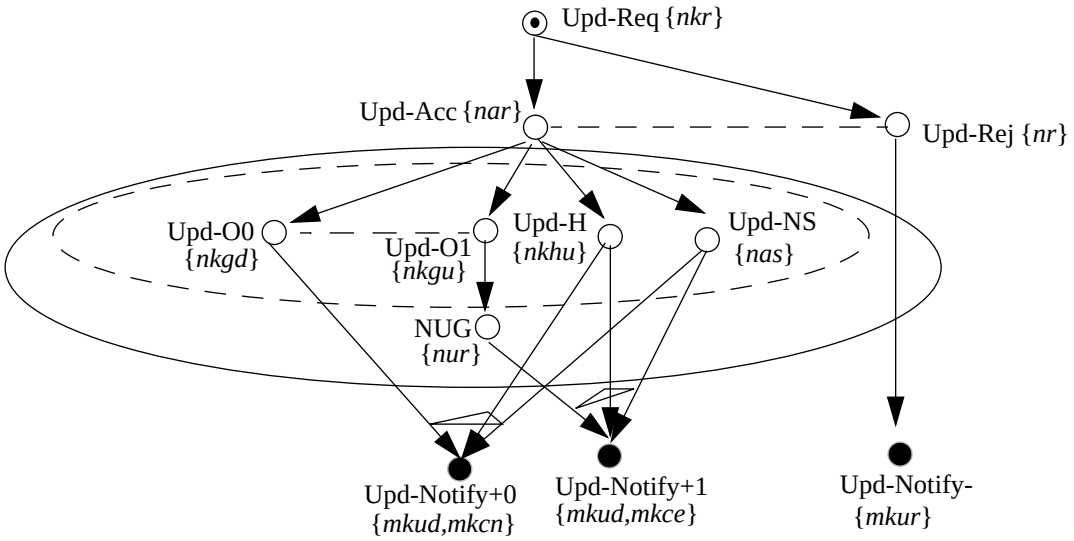


Figure 13. Event structure for routing update: step (iii)

Table 3: Events and postconditions in Fig. 13

event name	event	post cond name	postconditions
Upd-O0	*aGGSN deletes its record for aMS and aOldSGSN, and aNewSGSN is notified of this	*nkgd*	aNewSGSN knows aGGSN deleted its record for aMS and aOldSGSN
Upd-O1	*aGGSN updates its record for aMS and aNewSGSN, and aNewSGSN is notified of this	*nkgu*	aNewSGSN knows aGGSN updated its record for aMS and aNewSGSN
NUG	aNewSGSN adds the RoutingContext of aGGSN for aMS	*nur*	aNewSGSN added the RoutingContext of aGGSN for aMS
Upd-H	*aHLR updates the Location of aMS and notifies aNewSGSN	*nkhu*	aNewSGSN knows aHLR updated the Location of aMS
Upd-NS	*aNewSGSN adds SubscriberData for aMS	*nas*	aNewSGSN added SubscriberData for aMS

Step (iv)

Finally, we go into the details of Upd-O1, Upd-O0, Upd-H and Upd-NS. Due to lack of space, we only consider the refinement of Upd-O1. The refinements of Upd-O0, Upd-H and Upd-NS are similar.

The postcondition (*nkgu*) of Upd-O1 depends on the condition that *aGGSN updated its record for aMS and aNewSGSN (gu)*. We use GU1 (with postcondition *gu*) for the event that *aGGSN updates its record for aMS and aNewSGSN*.

To do GU1, aGGSN should know the update request, yet aNewSGSN does not know the address of aGGSN in order to inform it of the update request. However, by *general knowledge (A6)*, aMS knows the address of aOldSGSN, so we can let the update request from aMS to aNewSGSN to include the address of aOldSGSN, and thus, aNewSGSN knows the address of aOldSGSN. By this, together with *(A1)* and Figure 6, we know that aNewSGSN can talk to aOldSGSN and tell it the update request. By *(A5)*, on the other hand, aOldSGSN knows the address of aGGSN, so once aOldSGSN knows the update request, it can inform aGGSN about it. We use NTOU for the event by which aNewSGSN informs aOldSGSN of the update request, and OTGU for the event by which aOldSGSN informs aGGSN of the update request. Thus, we have the following events and their enabling relations:

NTOU {*okr*} --> OTGU {*gkr*} --> GU1 {*gu*}

Once aGGSN has updated its record for aMS and aNewSGSN, it can again ask aOldSGSN to pass this message to aNewSGSN. We use GTOU1 (with postcondition *okgu*) for the event that *aGGSN tells aOldSGSN it has updated its record for aMS and aNewSGSN*, and we use OTNU1 to name the event that *aOldSGSN tells aNewSGSN that aGGSN has updated its record for aMS and aNewSGSN*. Thus we have the following events and their enabling relations:

GU1 {*gu*} --> GTOU1 {*okgu*} --> OTNU1 {*nkgu*}

All together, we have got the way to obtain postcondition *nkgu*:

NTOU {*okr*} --> OTGU {*gkr*} --> GU1 {*gu*} --> GTOU1 {*okgu*} --> OTNU1 {*nkgu*}

The refinements of Upd-O0, Upd-H and Upd-NS are similar to this refinement of Upd-O1. The final event structure is shown in Figure 14 and Table 4 gives the meanings of the events and

postconditions that are refined on this step.

Table 4: Event and postconditions in Fig. 14

event name	event	post cond name	postconditions
NTOU	aNewSGSN tells aOldSGSN the update request	*okr*	aOldSGSN knows the update request
OTGU	aOldSGSN tells aGGSN the update request	*gkr*	aGGSN knows the update request
GU0	aGGSN deletes its record for aMS and aOldSGSN	*gd*	aGGSN deleted its record for aMS and aOldSGSN
GTOU0	aGGSN tells aOldSGSN it has deleted its record for aMS and aOldSGSN	*okgd*	aOldSGSN knows aGGSN deleted its record for aMS and aOldSGSN
OTNU0	aOldSGSN tells aNewSGSN that aGGSN has deleted its records for aMS and aOldSGSN	*nkgd*	aNewSGSN knows aGGSN has deleted its records for aMS and aOldSGSN
GU1	aGGSN updates its record of SGSN for aMS and aNewSGSN	*gu*	aGGSN updated its record for aMS and aNewSGSN
GTOU1	aGGSN tells aOldSGSN it has updated its record for aMS and aNewSGSN	*okgu*	aOldSGSN knows aGGSN updated its record for aMS and aNewSGSN
OTNU1	aOldSGSN tells aNewSGSN that aGGSN has updated its record for aMS and aNewSGSN	*nkgu*	aNewSGSN knows aGGSN updated its records for aMS and aNewSGSN
NTHU	aNewSGSN asks aHLR for Subscriber-Data and asks it to update its Location for aMS	*hkns* *hkr*	aHLR knows aNewSGSN needs to know Sub-scriberData of aMS; aHLR knows the request to update its Location for aMS
HU	aHLR updates its Location for aMS	*hul*	aHLR updated its Location for aMS
HTNU	aHLR tells aNewSGSN that it has updated its Location of aMS	*nkhu*	aNewSGSN knows aHLR updated its Location of aMS
HTNS	aHLR tells aNewSGSN the Subscriber-Data of aMS	*nks*	aNewSGSN knows the SubscriberData of aMS
NUS	aNewSGSN adds SubscriberData for aMS	*nas*	aNewSGSN added SubscriberData for aMS

Section 4. Use cases and scenarios in event structures

There is no agreed formal definition of use case or scenario in the literature. Hence, we propose to take our event structures as representing use cases. We shall now deal with the derivation of scenarios.

Let *S* be an event structure. A *system run* in S is any trace *r* of *S* that satisfies the following conditions:

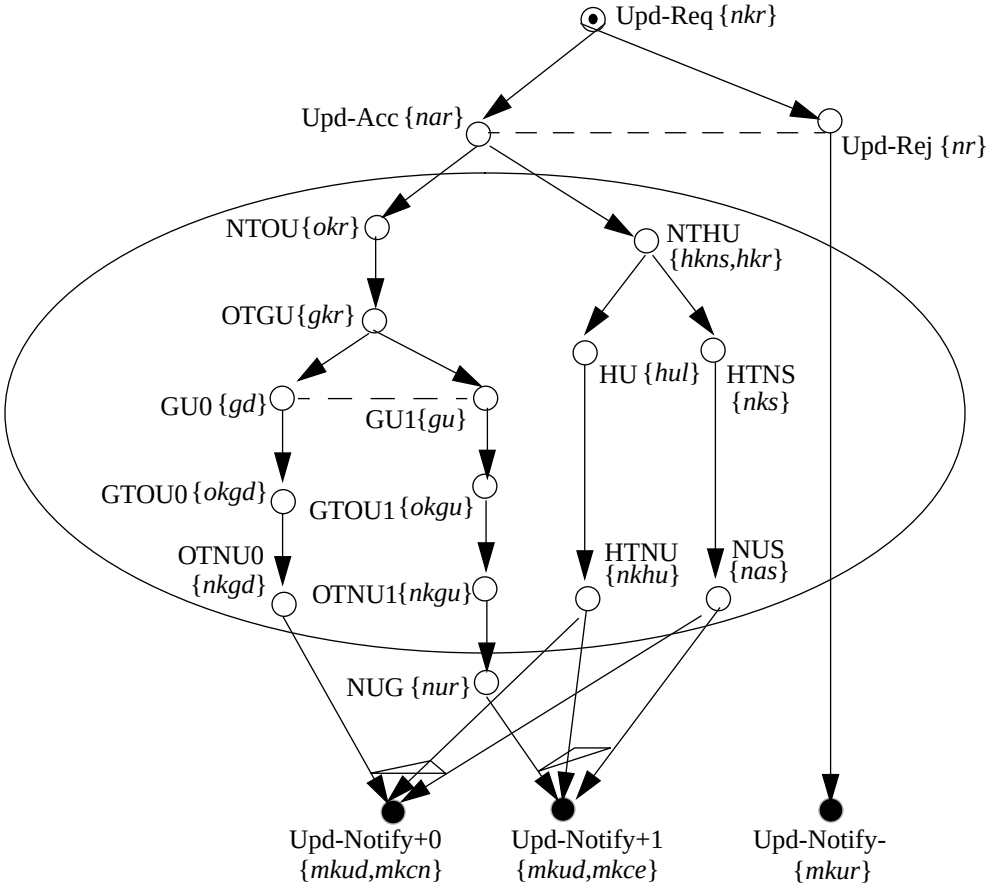


Figure 14. Event structure for routing update: step (iv)

1. *r* starts from an initial event and ends at a final event;
2. an event *e* in a run *r* is preceded in *r* by all the actions in one of its enabling sets (we assume for simplicity that an action can occur at most once in a run);
3. for any event *e* in *r*, no event in conflict with *e* in *S* appears in *r*.

In the previous section, we have seen how to obtain an event structure *S* from knowledge requirements *R*, by way of stepwise refinements. A *sequence of events* (*scenario*) satisfying *R* corresponds to a system run in *S*. So

1. to obtain a scenario from given knowledge requirements is equivalent to find a system run in the event structure obtained from these requirements;
2. to verify whether a scenario satisfies the given knowledge requirements is equivalent to verify whether it is a system run in the event structure obtained from these requirements.

Once we have obtained event structures from knowledge requirements, we can either obtain the scenarios that satisfy the requirements, or verify whether a given scenario satisfies the requirements.

One of the system runs derived from Figure 14 is

> Upd-Req, Upd-Acc, NTHU, HTNS, NUS, HU, HTNU, NTOU, OTGU, GU1, GTOU1, OTNU1, NUG, Upd-Notify+1

The meaning of this event sequence is illustrated in Figure 15. This figure shows the scenario

where aMS moves from location area LA1 (served by a SGSN called aOLDSGSN) to LA2 (served by aNewSGSN). aNewSGSN in LA2 detects the arrival of this aMS and proceeds to the routing update. This is expressed as the first event Upd-Req (1). aNewSGSN does the authentication check which we assume to be successful (internal event of aNewSGSN denoted by Upd-Acc (2)). aNewSGSN then asks aHLR for the information about this new aMS, information that is provided (3)(4). aNewSGSN adds the information and the aHLR updates its own information on aMS (5)(6). aHLR then informs aNewSGSN that the update has been completed (7). aNewSGSN then informs aOLDSGSN of the fact that it is now serving aMS (8), and this information is propagated to aGGSN, which then informs aOLDSGSN of the fact that the update is complete (9)(10)(11). aOLDSGSN informs aNewSGSN of the fact that these updates have been completed (12). This process ends with further updates in the aNewSGSN (13), and concludes with the notification to aMS that the process is completed (14).

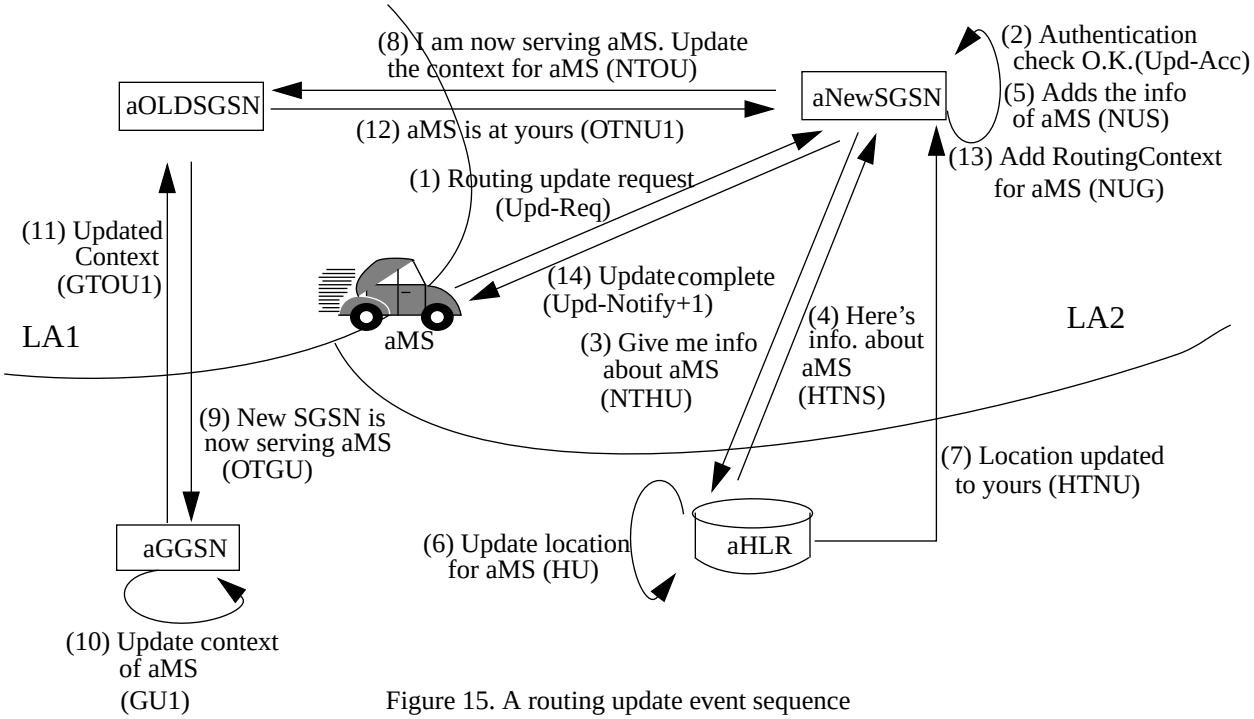


Figure 15. A routing update event sequence

In distributed system design, often one is interested only in the communications among the protocol entities, or agents. Internal events such as *HLR updates its Location for MS* are not of interest. This means that sometimes we need to ignore the internal events. By doing so in the above event sequence, we have the following scenario, illustrated in Figure 16 in the form of a Message Sequence Chart:

Upd-Req, NTHU, HTNS, HTNU, NTOU, OTGU, GTOU1, OTNU1, Upd-Notify+1

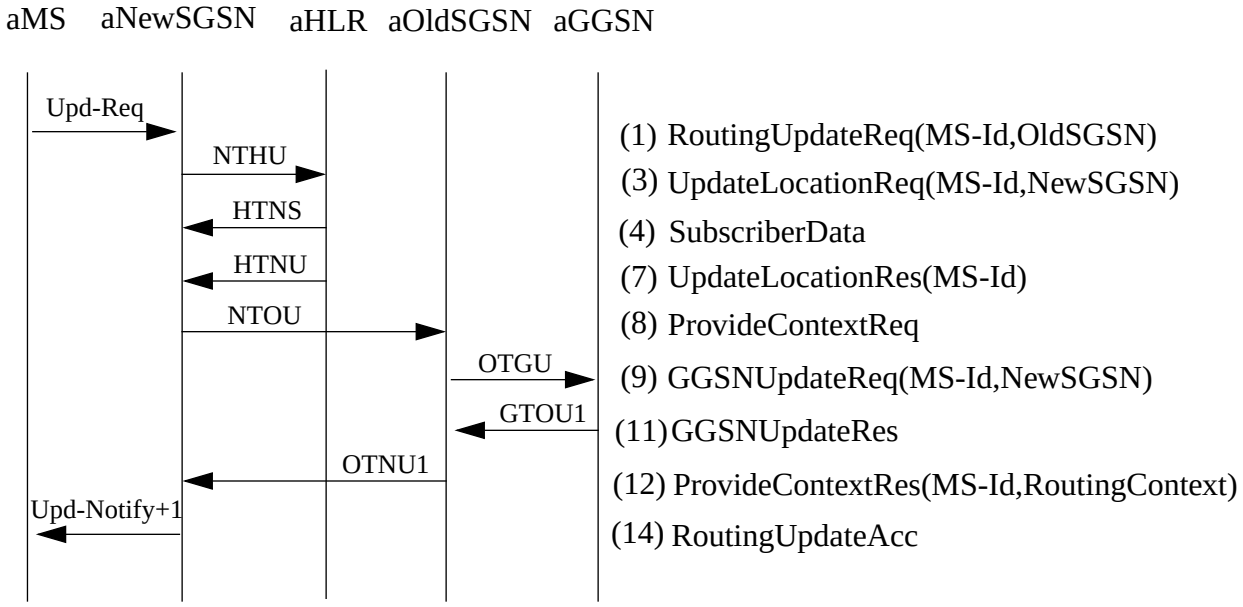


Figure 16. Routing update use case

5. Conclusions and final remarks

Starting from the idea that the function of a distributed system is to distribute knowledge, it was shown how the knowledge requirements of a distributed system can be specified. It was then shown how a partial system design can be developed by using these requirements, together with other facts, such as architectural constraints and other general knowledge about the system.

Event structures, corresponding to use cases, were used to express the design. Post-conditions were associated with events, and they must be shown to imply the desired knowledge requirements. Event structures can be refined progressively, and the correctness of each refinement can be verified by checking implication relations between postconditions. These event structures (at any stage of refinement) can be used to generate scenarios, which in turn can be used to check whether a design provides the desired system behavior, as further guidance to implementors, or as a basis for the generation of test cases. The method was demonstrated by using as example a mobile packet switching protocol.

This paper illustrates a way of designing distributed systems that is natural and intuitively used, but has not been developed in the literature. In a theoretical setting, it appears to be an application of the *proof searching* concept that is advocated by many authors as a method for generating correct programs from requirements.

There are several topics for further research, one of them being the formalization of the method. The reader will have noticed that we have used many English expressions that could be translated to precise statements in logic, but at the cost of many definitions. Further, the results of our method are not complete designs, because at this point we do not specify recursion or iteration.

Use Case Maps (UCMs) [BuC95, ALBG99] are a use case notation that is gaining rapid acceptance in industry. It includes concepts of pre- and post-conditions. We are planning to adapt the ideas of this paper to the UCM notation.

We are also at the early stages of developing a tool to support this design approach. This

tool (being implemented in Java) has the goal of helping in the routine tasks of generating event structures, preconditions, and postconditions. So far, it takes postconditions as they are written, without analyzing them. Eventually, it would help in formalizing requirements from statements written in a stylized form of English, as well as checking consistency of refinements.

A further topic of interest is the development of a testing theory based on the satisfaction of logic requirements. In our example, one may wish to test an implementation for being able to establish a logical link to aNewSGSN. By tracing scenarios leading to this postcondition, appropriate test cases can be derived.

Acknowledgments. This research was completed while the first author was at the University of Ottawa under a postdoctoral fellowship. It was funded in part by grants from Motorola Canada (Advanced Radiodata Research Center), and the Natural Science and Engineering Research Council of Canada. We thank Brahim Ghribi for the information he provided about GPRS, and for many useful discussions. Rossana de Castro Andrade provided a number of useful comments.

References

[AL93] M. Abadi, L. Lamport. Composing Specifications. ACM Transactions on Programming Languages and Systems, 15(1):73--132, 1993.

[ALBG99] Amyot, D., Logrippo, L., Buhr, R.J.A., Gray, T. Use Case Maps for the Capture and Validation of Distributed Systems Requirements. Fourth IEEE International Symposium on Requirements Engineering (RE'99). Limerick (Ireland), 1999, 44-53.

[BoCa89] G. Boudol, I. Castellani. Flow Models of Distributed Computations: Event Structures and Nets. Technical Reports INRIA, Sophia Antipolis, 1991.

[BuC95] R. J. A. Buhr, R.S. Casselman. A Use Case Map Approach to High Level Design of Object Oriented Systems. Prentice-Hall, 1996.

[ETSI98] General Packet Radio Service (Draft). European Telecommunications Standards Institute, 1998.

[Hal87] J.Y. Halpern. Using Reasoning about Knowledge to Analyze Distributed Systems. Ann. Rev. Comp. Sci., No. 2, pp. 37--68, 1987.

[HF89] J.Y. Halpern, R. Fagin. Modeling Knowledge and Action in Distributed Systems. Distributed Computing, 3:159--177, 1989.

[Hin62] J. Hintikka. *Knowledge and Belief*. Cornell University Press, 1962.

[Jac92] I. Jacobson et al. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

[La91] R. Langerak. Bundle Event Structures: A Non-Interleaving Semantics for LOTOS. In: M. Diaz and R. Groz (Eds.) *Formal Description Techniques, V.* North Holland. 331-346, 1991.

[MP92] M. Mouly, M. P. Pautet. The GSM System for Mobile Communications. Published by the authors, 1992.

[Wi80] G. Winskel. Events in Computation. PhD Thesis, CST-10-80, University of Edinburgh, 1980.

[Wi89] G. Winskel. An Introduction to Event Structures. LNCS 354, pp. 364- 397, Springer-Verlag, 1989.