Language Support for Changeable Large Real Time Systems

Ivar Jacobson

Ericsson Telecom or Swedish Institute of Computer Science
S-12625 Stockholm Box 1263
Sweden S-16313 Spånga, Sweden

A set of concepts for modeling large real time systems is discussed informally. The concepts support the design of centralized as well as distributed systems. They are object oriented in that they correspond to entities of the 'real world', and they are 'change oriented' in that they support not only the first development stage of a system but also its continuous change and evolution. In particularly, the concepts give a promising solution to 'on the fly' changes of existing, active entities.

1. INTRODUCTION

The work presented here was initiated in 1967, when a new set of concepts was proposed for the development of large telecommunication systems. The concepts were soon followed by a skeleton of a new design method, the use of which was first demonstrated in the development of the AKE system put in service in Rotterdam in 1971.

In 1967 Ericsson had long traditions in the development and production of large electromechanical telecommunication systems. The original proposal can simply be summarized as an attempt to unify long experience from systems design with the possibilities offered by a dramatically new technology - computer technology. Since the two technologies were so different, this was not a self-evident approach, neither within Ericsson nor within computer science. There was rather a strong attitude that the two represented unrelated, technological universes: the new one was so different that it would be meaningless and only burdening to make any attempt to learn from the old one.

2. THE EARLY CONCEPTS

A real time **system** is an open system communicating with its environment by signals only. A **signal** models the physical stimulus/response communication which a concrete system has when interacting with the outside world . Signals are typically directed one-way with a 'send/no-wait' semantics. Each signal has a unique name and carries a sequence of data objects, e.g. the signal 'digit: value', which corresponds to the stimulus sent when a subscriber dials a digit (with a particular value) in a telephone number.

A system offers, or is capable of performing, a set of services (the Ericsson terminology is functions). Given an input signal, a system performs internal actions such as executing algorithms, accessing internal information, storing results, and sending output signals to the environment.
The model just described represents one view of a system - the most abstract view: the system as a black box. We call this view 'the centralized view' to be distinguished from 'the distributed view', which models the system as a set of interconnected **blocks** (not to be confused with blocks as used in sequential programming languages). Blocks are modules which can be implemented in hardware or software or any combination of both. A block communicates with its environment only through signals. Signals between two blocks are internal, whereas signals modeling physical communication, i.e. signals between a block and the environment of the system, are external. Internal signals are messengers conveying data from one block to another within the same system. In the original suggestion, an internal signal between two software blocks was proposed to be implemented as a subroutine call (the sending action) from one block to a labeled statement (the receiving action) representing the beginning of the subroutine. (Recall that the original proposal was given when programming real time systems in assembly languages.)

Now, given the distributed view of a system, the system is a set of interconnected blocks jointly offering the services of the system. Each block has a program which it obeys on reception of an input signal, performing internal actions (i.e. executing algorithms, storing and accessing block internal information), and sending internal and external signals to the environment.

The acceptance of the proposal was based on an intuitive understanding of some very important features of blocks and signals:

1) Blocks are manageable units for the design, production, installation, operation, maintenance, etc. of large systems. This is a software engineering aspect of blocks, making possible the division of the work on a large system into parts that can be planned, worked, tested, produced,..., separately and then integrated as a system.

2) Blocks are units of encapsulation. The only means of accessing the internals of a block is through a strictly standardized signal protocol. Only the signal protocol is visible from outside the block, the internal structure and implementation being hidden to the user of the block.

3) Signals offer dynamic interconnection of blocks, for instance a given block A can be interconnected to many different other blocks, but in a given situation the receiver block B is dynamically known to A as a data object. When

A sends a signal to B, the actions taken is decided solely by the receiver block B, and the sender block A specifies only the signal name and the data object referring to B.

Given a signal the receiver decides, entirely at run-time, what code to execute.

4) Signal communication means that the sender does not wait for the receiver to be ready but continues with other actions. The receiver can, when busy, queue the received signal. The sender and the receiver work as two autonomous sub-systems and do not synchronize their actions. Thus the risk of getting dead-locks is very small.

5) Blocks can be implemented using different techniques for different blocks, e.g. different programming languages, different computers or computer systems, different hardware techniques, etc. Block decomposition therefore supports adaptation to new technology without redesign (or with delimited redesign).

The 2nd point is an important feature of data abstraction [15], and the 3rd is an important feature which is called dynamic binding in Smalltalk 80 ([7] and [20]) and similar languages such as Objective-C [4].

Decomposition was guided by the following rules.

a) Service modularity or 'one service - one block' is the primary objective of the decomposition. This principle is especially easy to apply to optional services such as operation and maintenance services.

However, a strict service modularity (one-to-one relation) can only rarely be achieved since services normally require access to the information of one another. Such accesses require a (small) part of the accessing service to be 'allocated' to the block having the other service, and signalling to and from that part.

b) Each type of input/output device is encapsulated by a block, for instance an incoming trunk or an outgoing trunk line is enclosed by a block. Apart from the hardware of the physical device, an input/output block also encloses the (main) service defining the communication protocol of the line connected to the device, and all parts of other services requiring access to data objects of the main service. An input/output block in AKE contains typically about 25 parts of services (primarily operation and maintenance services) other than the one defining the protocol.

c) A data base should be enclosed by a block and thereby only be accessible through signals to the block. Examples are the analysis tree required to find the desired direction of a call, or the data base for the switching network keeping track of busy lines and busy interconnection paths.

Essentially the system modeling concepts and the design method outlined in the late sixties are still in use in the design of the AXE system which is the successor of the AKE system. The main differences, and these are indeed very important [9], are that not only external signals but also internal signals have a 'send/no-wait' semantics, that the program design activities use a programming language that features this signal communication, and that signalling has been implemented through micro-programming with a technique that characterizes capability-based computer systems (or object oriented architectures) [14]. The design method and its careful implementation have played an important role for the AXE-system. The key feature of the system is its general manageability over all related processes: marketing, projecting, design, production, installation, operation and maintenance. The system is considered remarkably easy to change: it supports functional evolution and adaptation to new technique.

Apart from the AKE and the AXE systems, the concepts have also been used for the MD110 system which is a digital PABX, and for the AXB system which is a telex and data switching system.

3. THE NEW CONCEPTS

Based on our experience from the AKE/AXE systems we naturally would like to provide constructs to further decompose a block into lower level blocks, which in their turn could be decomposed...etc., until the lowest level blocks corresponded to primitives of some types. Unfortunately, signal communication, intended to model physical signaling with 'send/no-wait' semantics, is too explicit and does not offer the desired abstraction. Instead another model was found, the object/message model used in object oriented programming (see survey in [19]), very close to our own block/signal paradigm. The two design techniques, developed independently and in parallel, have notably much in common but they solve basically two different problems - 'large scale' and 'small scale' systems. Blocks were only intended for the more abstract entities of the system and primarily developed in a top-down fashion from the requirements of an application. Objects were primarily (but with exceptions) developed to provide programmers with a tool-box used to meet the requirements from an application; new objects were designed on top of the existing ones.

We now give an intuitive description of the new concepts developed since 1978 ([11] and [12]*). The concepts have been concretized in the form of an example language FDL ([12] and [13]). FDL is a programming language formally defined in VDM ([1] and [2]) extended with CSP like constructs [10] for specifying concurrency.

For our previous concepts, the block concept was the cornerstone serving the combined purpose of supporting three distinct structuring needs: to represent autonomous sub-systems, to provide a service oriented structure, and to handle data objects. The new concepts support these three needs by three structural components reflecting the object orientation of the model, and they provide two methods for communicating among them. The structural components are three kinds of classes (or types): **blocks**, **services**, and **objects**, all three with dynamically created **instances**. The two communication methods are **signals** and **messages**. From earlier we will recognize blocks and signals, although as will be described, blocks now have a different role in comparison with their earlier counterparts.

The new modeling concepts will be presented in a 'bottom-up' order as may be preferred by readers with some language design experience, and only when required by an unknown context will we distinguish a class such as an object from instances of that class, i.e. instances of that object.

Objects

Objects are models of 'real world' entities manipulated by services (or other objects) in the system.

Some objects are primitive and similar to 'built-in' data types, e.g. integers, Booleans, etc. The other, non-primitive objects are defined by designers; these objects respond to messages from services and other objects and may refer to other objects,

* The research has been reported as a Ph. D. thesis [12] developed within Ericsson, Stockholm, in cooperation with the Department of Computer Systems of the Royal Institute of Technology in Stockholm.

primitive or non-primitive, abstract objects.

Each abstract object possesses a data item indicating a current node in a state transition graph (fig. 3) describing its overall behavior. An action on an object is represented by a transition between two state nodes in the graph. The transition is a sequence of actions on other objects - primitive or abstract (fig.1).

Messages

Actions on an object are invoked by receipt of messages appropriate to the object's current state. Every action is acknowledged by a reply message to its invoker, possibly carrying result data. Hence a 'remote procedure call' [17] effect occurs, with a state dependent selection of the appropriate action, i.e. transition, to apply.

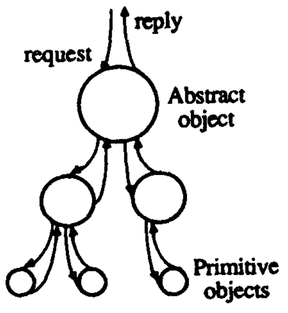

Fig.1 Objects and messages.

The sender of a message waits until the matching reply message is received before completing its own transition. This is termed 'send/wait' semantics.

A message synchronizes its sender and receiver, but it also is a carrier of 'data' known to the sender and, after the communication act, also 'known' to the receiver. These 'data' are objects or services - instances as well as classes.

Signals

From this perspective - as a carrier of 'data', a signal is similar to a message. However, a signal is directed one way, thus no reply message is required from the receiver.

Furthermore, signals obey the 'send/no-wait' semantics described previously. To achieve this semantics the sender and the receiver, which are service instances, have an associated port. A port contains a queue of received but not yet treated signals and another queue of output signals not yet sent.

Signals are used for communication external to a system. However, signals can be used within a system as well. Such internal use of signals is quite appropriate when the coupling between the sender and the receiver must not be synchronized in any direct manner. For our concepts, such communication is the basis for decomposing a system into blocks resulting in distributed systems.

Services

A system offers or is capable of performing a set of services.

Services model the features of a system. Moreover, intuitively:

Some services are directly connected to an external user, such as a subscriber, an operator, a maintenance person, etc. This external user participates in a dialogue with the service, triggering the step-by-step actions required to accomplish the desired service.

Other services participate through another service in the dialogue with an already connected user. Such services represent an extended service to the user, e.g. the abbreviated dialling service.

Furthermore there are services, such as telephone calls, that serve the combined purpose of being a communicating partner to one user and an extended service to another. In this case the two users participate in one way or the other in the same course of events.

Statically a service encloses or possesses a set of abstract objects. This 'possess' is used to facilitate putting together a system as a set of services. We configure a system, i.e. a block, by specifying the services only and do not need to specify any lower level objects, since these always follow their service. This 'possess' has nothing to do with how objects are used dynamically which soon will be discussed.

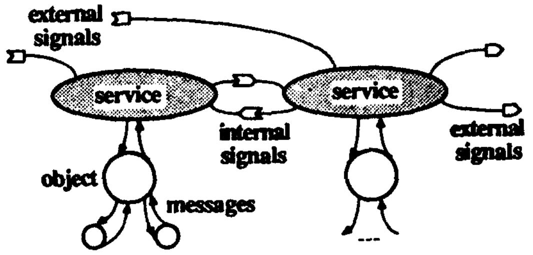

Fig.2 Services and signals.

An input signal to the system triggers (fig.2) one of the services, which obeying its state transition graph (fig.3) performs actions on objects and, in its own turn, triggers other services by sending output signals. Some of these signals cause the connection (or disconnection) of an extended service, whereas others trigger an already connected service.

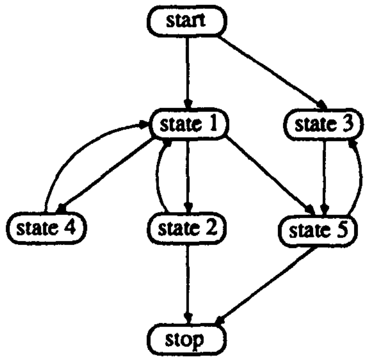

Fig.3 A state transition graph.

A graph specifies the transition of a service or an object given an input signal or a request message, respectively. A transition of a service causes actions on objects to which the service sends messages (and receives replies), and the delivery of output signals to other services. A transition of an object is restricted to actions on other objects and no signalling is allowed.

Given an external input signal, the behaviour of the system is deterministic. The actions performed by services are atomic, i.e. logically instantaneous [8]. Hence, even if executed with a high degree of concurrency (the usual case), the actions of the set of service instances involved in a course of events always appear to have been done in some overall sequential order. This serialization is applied to the reception of input signals with internal signals having a higher priority than external. Implementation ideas of atomicity, particularly inspired by [18], are discussed in [12].

A course of events, such as a telephone call, is an important entity of the 'real world' which we model by a special semantic construct simply called **course**. A course represents the set of service instances interacting among themselves and with an external user or a group of related external users. In an implementation the consequences of a failure can now be delimited to a course of events and does not require more severe recovery actions. Moreover, since functional changes can be made by changing each course of events atomically, the course concept simplifies making functional changes 'on the fly'.

In concrete terms, services may be thought of as generalized objects that only communicate through signals with other services - external or internal to the system. However, data manipulated by services are generally separated into individual objects, e.g. for reasons of increased modularity and representation hiding. Objects play a totally internal role in system structuring, while services provide external, application oriented interfaces to them. Thus, to the external user, services constitute the logical structure of the system, internally manipulating objects which are invisible to the user for clarity and system evolvability.

Blocks

Blocks model autonomous sub-systems, communicating asynchronously with their environment. Compared with the earlier block concept, blocks now statically have a limited role: to model sub-systems, but not services or data bases. Dynamically blocks serve as serializers for input signals to its services.

Actually, a system as in the preceding sections is a block. For instance, a telecommunication system is a block and a corresponding telephone exchange is an instance of that block.

Therefore:

A block offers or contains a set of services (fig.4). The communication between the block and its environment is carried out by its services, which as is known only communicate by using signals. Statically, a service contains a set of service associated objects (fig.4) which are accessible from other services within the same block, but none is accessible to an external user.

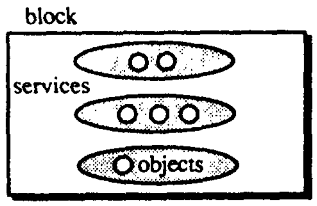

Fig.4 The static, centralized view of a block.

Dynamically, a service (a service instance) participates in a dialogue with an external user during which the service performs actions on the objects and communicates with signals with its user or with other services - external or internal to the system. Actions are performed by sending and receiving messages to objects.

'Data' carried by a message or a signal are references to objects or services, classes as well as instances, known to the sender. Accesses to primitive objects are allowed since these objects are globally known. Accesses to abstract objects or services are restricted; the accessing one and the accessed one must both be contained by the same block. However a reference to an object or a service can without restrictions be used as 'data' in a forthcoming signal.

We have now presented the centralized view of the block. Another view - the distributed view - models the block as a set of interconnected blocks. A block can be decomposed into sub-blocks (fig.5) which express a plan for the physical structure of a particular system implementation. Such decompositions are pragmatically crucial but semantically transparent. These 'lower level' blocks can also be modeled from the two points of view - the centralized and the distributed view.

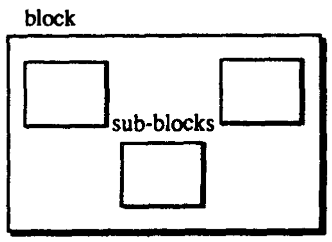

Fig.5 The static, distributed view of a block.

A service of the 'top level' block is implemented by some of the lower level blocks in cooperation. Ideally, a service is implemented by one block only. However, since services can access abstract objects of one another but an object can be allocated to one block only, the larger services must be partitioned over two or more blocks. In such cases the service is partitioned into a number of service parts that are allocated to exactly one block each and then, being a service of that block, communicating with its 'siblings' through signals. Service parts are located in blocks in such a manner that if they deal with a particular object, they will as far as possible be allocated to the block possessing that object.

Given the action atomicity and the decomposition transparency requirements above, it suffices for semantic purposes to define the effect of a single service responding to a single signal. This semantic 'factorability', by the way, is an indication of the simplified computational domain within which designers will operate: each will be permitted the logical luxury of believing he or she is the sole possessor of a stable data environment. Of course there is much detail involved in the careful delineation of all the steps in such an overall action.

Class Inheritance

Blocks, services and objects are structured in three different class hierarchies, let alone with a common root. A new class can be described in terms of an older, usually more general class through a technique generally called inheritance [20]. The new class (a sub-class) is specified by describing how it differs from the older class. In this hierarchy, lower level classes inherit properties of higher level classes. A new block can be designed as a sub-class of another block and can be given some new or some changed services compared with that block. A new service can be based on an existing service but with an extended signal protocol, etc.

5. SYSTEM DEVELOPMENT

The separation of individual services can be maintained throughout the specification and the design process. When changes in existing services are required or when new services are introduced, their specifications can easily be mapped into modifications of the existing system documentation. Therefore, this modularity is intended to be retained in current system implementations, thereby increasing the general system manageability.

A system specification, possibly narrative, is the input to the system design work (fig.6). We require from this document that each service should be factored out and specified separately in a service specification. What constitutes a service is not primarily a design decision, but a decision taken from a more general management point of view, i.e. that services are the handling units of the many important activities mentioned previously.

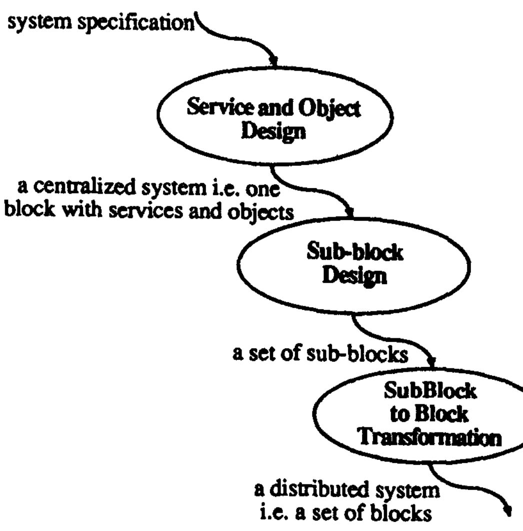
system specification
Service and Object Design
a centralized system i.e. one block with services and objects
Sub-block Design
a set of sub-blocks
SubBlock to Block Transformation
a distributed system i.e. a set of blocks

Fig.6 The system development steps using FDL.

1) Each specified service is directly, with a one to one mapping, implemented by an FDL-service and its set of abstract objects. The FDL system can be verified and validated against the system specification.

There are occasions, however, when a system must be implemented as a distributed system in the form of loosely connected autonomous blocks, for instance a switching system that must be divided into a subscriber sub-system and a group selector sub-system. In such cases, the design process will be divided into two more steps:

2) First, the system is decomposed into the requested set of sub-blocks, and the services and objects are allocated to these sub-blocks. Objects are allocated as a whole, whereas services may first need partitioning. The partitioning of services into service parts and the allocation of the parts to the sub-blocks can, due to the support given by FDL, be done automatically or with very little designer assistance. Note that top level transitions will still be performed atomically even if several sub-blocks participate and communicate through signals.

3) Next, we must loosen the interaction between the sub-blocks and not require transitions with two or more participating sub-blocks to be atomic. Therefore, each sub-block is transformed into an autonomous sub-system, i.e. a block 'loosely' connected to the rest of the system. Since sub-blocks already communicate with signals, the designer's task is to take care of the problems that arise from changing the scope of atomicity. Top level transitions are not atomic over different autonomous blocks, only between the sub-blocks within a block. However, this task can be greatly simplified by an appropriate tool.

The use of a formal language similar to FDL, fully supported by tools, will result in systems with only a few number of blocks, considerably fewer than was advocated by the early design method. Services and objects are expected to replace and even highly improve the modularity, which earlier only could be given by blocks.

6. FUNCTIONAL EVOLUTION

Large systems will be subjected to continuous changes during their entire lifetime. These changes originate both from requirements of new or changed services and from incorporation of new hardware techniques.

In an object oriented system, only two kinds of items exist and are relevant for changes: object classes and object instances. This is due to the uniformity of the object world. Furthermore, names in this world have the same form regardless whether they refer to a primitive object instance or to an abstract object instance.

The introduction of new classes followed by the creation of new instances is easily explained in object oriented systems. Such changes are naturally offered by these systems.

Apart from object classes and object instances, an FDL-system is inhabited by some more items: block instances, services, service instances and courses. A change in a system has impact on almost all activities, two of the more important are the system development activities and the operation and maintenance activities of an installed system.

System development

Underlining the change orientation of the methodology, we view the system development as a change activity, changing an existing system into a new system. Thus, the first development cycle is a change from 'nothing' to 'something' (see [12]).

Since the service concept directly models features or functions, the design method supports functional evolution in a straightforward manner. In some cases a new service can be designed without changing existing services. This case includes a large number of functional changes due to the dynamic binding of signals. This category also includes changes where a new service only requires accesses to the existing objects of another service. In other cases a new service needs to be invoked by an existing service. Such changes, not required by the existing service itself but required to introduce a new service, should be defined in association with the new service. In the remaining cases, an existing service must comply with new requirements, and thus its definition or, if a minor change, an object definition only must be altered. Such a change will be made as an increment, specifying what is removed and what is added, before integrating the existing definition with the increment into a resulting definition.

Operation and maintenance

The first functional change of a concrete system is the installation of a that system (change from 'nothing' to 'something'). First a new block (instance) is created to represent the new system. The created block is originally 'empty', but will be changed by 'filling' it with behaviour. If the block is undecomposed, it will be 'filled' by creating a new service class for each new, required service, and then each service is given its expected behaviour. For each abstract object contained by the service, a new object class is created and given its specified behaviour. If the block is decomposed, it will be 'filled' by creating instances of its lower level blocks, and then these are 'filled' as a block in general is 'filled' with the service parts making up the services of the block.

A subsequent functional change should be brought into the system as an atomic action not 'disturbing' the externally visible behaviour of the system. The following types of changes must be covered by our method: (i) a new service is incorporated, not requiring any change of the existing services, (ii) an existing service or any of its objects must be changed, but existing instances can follow the behaviour specified by the old class, (iii) as the second case, but the existing instances must be transformed into new instances following their new classes. This is a most difficult problem.

Therefore we firmly believe that constructs must be provided to precisely define the semantics of transforming an existing version of a block instance into a new version of the same block instance. We will in the following carefully describe the support for functional evolution offered by our concepts. At present, however, the formal definition of FDL does not incorporate the full extended change facility - cases (ii) and (iii) are not yet covered.

Functional evolution is of two types:

Extensions of existing behaviour, by which we mean that new services are introduced but the existing services are not changed. These changes are called *functional extensions*.

Existing services are changed, which we simply call *functional changes*. A new version of an existing service is installed.

Functional Extensions

Let us refer to an existing set of services with their objects by the invented word '*existion*', and refer to the extended set of services as an extension. There are only two kinds of relations between an existion and an extension.

The extension requires no accesses or read accesses only to an existion.

The extension requires control access (i.e. the extension causes additional instructions to be executed) from an existion, without changing the existion.

The first case means that the extension can use existing actions on an object instance provided these actions do not change the state of the object instance. This kind of changes is directly supported by the service concept of FDL. A feature or a function can be directly mapped onto an FDL-service.

The second case means that while an existion is executed atomically, the extension may 'intervene' at specified points. When the extension is executed, the control will be returned to the existion which now continues its atomic action. More than one extension may intervene in the execution of an existion. A *probe* specifies where the intervention is required in the execution of an existion.

We must provide such constructs so that the extension can be described without changing the existion, i.e. the existing service and object definitions. They must not be changed, since they describe well working existing services which in this case must not change. In fact, our design methodology advocates that also the initial system design work makes use of this technique for extensions when describing optional services. Other languages, such as Ada [3], CHILL [6], CLU [16], .., but also Smalltalk-80 [20] require that the second kind of extensions make changes in existing, well working programs. The result is often very messy after some extensions.

The idea is to provide an extension service (or object) with a list of probes (fig.7). A probe specifies an insertion point in the graph of the extended service (or object). During interpretation of a transition path, a service instance or an object instance in the existion allows the desired statements of the extension (service or object) to intervene.

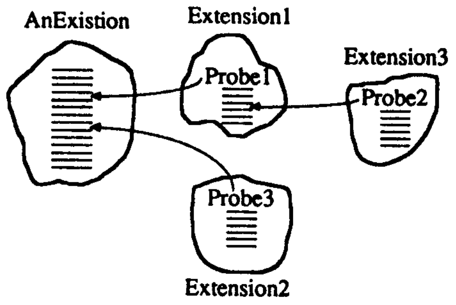

Fig.7 Functional extensions.

An extension may itself be treated as an existion and be intervened by another extension.

Since functional extensions do not change the behaviour of existing services, these changes can be introduced in a single step.

Linguistically, an extension can at first glance be viewed as a new class inheriting its existion. However, there is a very important difference. The normal class inheritance mechanism allows the construction of a completely new service by using an older class. This new service only uses the older class to specify the behaviour of its instances. Instances of the new service and instances of the old one may exist independently of one another and they can execute independently of one another in a system. For instance, a Call service can in this way be used to define a more specialised service, for simplicity called Call+. Now, the Call service and the Call+ service can be installed and executed independently.

However, class inheritance is not the phenomenon desired here. Instead, an extension must exist together with its existion; it always requires its existion to be installed and it will only be executed when its existion is executed.

Functional Changes

Here an existion is to be transformed into a new existion (fig.8). The intention is to change the functional behaviour of existing services, retaining the same names. The new existion can be viewed as a set of classes of normal type, inheriting the behaviour of the older existion which is to be replaced.

In an active system, old existions must be transformed into new existions.

1) As long as the change is not retroactive (existing instances obey old classes), this is a simple problem. The existing class coexists with the new class until no instances of the existing class are alive. The meaning of a name must be extended. Instead of one component, it should consist of two components: the 'new' name and a version number. A

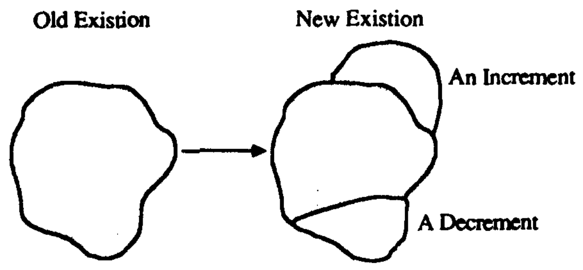

Fig.8 Functional changes.

definition with a given name can exist in several versions; only the last one is available for new installations. Formally, the 0th version is 'empty'. The 1st version defines the change from an empty definition to one that defines a released behaviour. The 2nd version defines an 'approved' change from the 1st version. Each new version is described as an increment to and a decrement from the previous one (fig.8). When a new version has been installed successfully, its definition can be integrated with the previous one to a new definition which is efficient for interpretation and which can be used for subsequent changes.

When two classes with the same name are in operation simultaneously, all new creations are directed to the new class. For instances with a short lifetime, the change will be in power when all 'old' instances have been removed from the system. For long-living instances, more powerful means must be used:

2) The problem is much worse if existing instances must also be changed to obey the new class. The solution to this problem is supported by the course concept of FDL. In an FDL system the computational objects are instances of services and objects within a block instance. Since the ongoing, independent activities correspond uniquely to courses, it is sufficient to change in one single step what is involved in a course.

To change a service instance or an object instance is to change its class reference, its state and its variable references. Due to the uniform reference mechanism, the semantics of a change performed as an atomic action is seemingly easy to formalize.

For other conventional languages this would be a very hard task since this simple mapping relation is not supported. Changes must often be carried out simultaneously on several objects. It is therefore very difficult to know exactly at any given moment on which objects computation is carried out. In languages like CHILL or Argus [17], the ongoing activities are performed by processes (or guardians), but normally several processes (guardians) jointly implement a course of event. Therefore, simultaneous changes have to be made in several processes (guardians).

6. Conclusions

From a general computer science perspective, FDL combines several techniques currently believed to be useful for system specification, design and implementation. These techniques include object orientation, message based control, and atomic actions.

In addition, FDL contains several new ideas which include:

The role of services for the logical structuring of real time systems corresponding to a feature or a function,

The use of messages and signals in combination,

The organization of an ongoing user activity as a course, reducing the consequences of a failure and facilitating functional evolution,

The idea of transparent service decomposition over sub-blocks,

The modeling of a system from two related aspects: the centalized view and the distributed view,

The inclusion of source language features for system evolution offering change orientation.

The new concepts are not only theoretically sound but, most important, they can rely upon the long experience from the use of similar concepts for the development, operation and maintenance of several large real time systems. Every single design decision can be evaluated, at least through common sense reasoning, against the firm experience from the whole life cycle of several systems.

Therefore, the new modeling concepts concretized in the form of a method supported by appropriate tools should result in a significant reduction of the system life time costs. But, most important, the customers of FDL designed systems should discover that it is remarkably simple to perform functional extensions and changes.

The object orientation of the language allows the different system life time activities to deal with 'things' corresponding to real world entities. For instance, our services correspond to real world features or functions, service instances to uses of a feature or a function, objects to externally perceived device types of different kinds, etc. This simplifies the total system handling activities by not requiring complicated translations between internal and external entities.

As a result, services corresponding to features or functions will be the basic handling unit for not only design and implementation but also for marketing, production, installation, operation and maintenance.

Blocks, first of all having the role to model asynchronous sub-systems, will be composed products used to package services into manageable sub-systems.

Objects, modeling device types, etc., will be lower level, reusable components in the design of new services.

The early and new concepts have not only had an impact on several systems developed within Ericsson, but they have also inspired some of the more important language features of the CCITT Specification and Description Language SDL [5]. This circumstance has encouraged and convinced us about the common usefulness of the model and its accompanying method for large real time systems.

ACKNOWLEDGEMENTS

The author is indepted to the L.M. Ericsson Telephone Co. for funding this research. He is also very grateful to professor Lars-Erik Thorelli for his personal and professional support and for his constructive criticism of the work.

REFERENCES

[1] D. Bjørner & C. B. Jones, The Vienna Development Method: The Meta-Language, Springer-Verlag, 1978.

[2] D. Bjørner & P. Folkjär, A Formal Model of a Generalized CSP-like Language, Proceedings of IFIP 80, 1980.

[3] G. Booch, Software Engineering with Ada, The Benjamin/Cummings Publishing Company Inc., 1983.

[4] B. Cox, Message/Object Programming: An Evolutionary Change in Programming Technology, IEEE Software, Vol.1, No. 1, Jan 1984.

[5] C.C.I.T.T., Fascicle vi.11, Functional Specification and Description Language (SDL), Rec. z.100-z.104, Geneva, 1984.

[6] C.C.I.T.T., Fascicle vi.12, CHILL, Rec. z.200, Geneva, 1984.

[7] A. Goldberg & D. Robson, Smalltalk-80 The Language and its Implementation, Adison-Wesley Publishing Company, 1983.

[8] J. Gray, The Transaction Concept: Virtues and Limitations, IEEE Proceedings of the Seventh International Conference on Very Large Data Bases, Sept 1981.

[9] G. Hemdal, AXE 10 - Software Structure and Features, Ericsson Review 53, 1976.

[10] C. A. R. Hoare, Communicating Sequential Processes, Comm. of ACM, Vol. 21, No. 8, Aug 1978.

[11] I. Jacobson, On the Development of an Experience-based Specification and Description Language, IEE Proceedings of Software Engineering for Telecommunication Switching Systems, July 1983.

[12] I. Jacobson, Concepts for Modeling Large real Time Systems, Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985.

[13] I. Jacobson, FDL: A Language for Designing Large Real Time Systems, Proceedings of IFIP 86, Sept. 1986.

[14] H. Levy, Capability-Based Computer Systems, Digital Press, 1984.

[15] B. Liskov & N. Zilles, Specification Techniques for Data Abstractions, IEEE, Trans. on Software Engineering, March 1975.

[16] B. Liskov, et. al., CLU Reference Manual, MIT-TR225, Oct 1979.

[17] B. Liskov & R. Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, Massachusetts Institute of Technology, ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983.

[18] D. Reed, Implementing Atomic Actions on Decentralized Data, ACM Transactions on Computer Systems, Vol. 1, No. 1, Feb 1983.

[19] T. Rentsch, Object Oriented Programming, SIGPLAN Notices, Vol. 17, No. 9, 1982.

[20] The Xerox Research Learning Group, The Smalltalk-80 System, Byte, Aug 1981.