통신 프로토콜 명세 언어 SDL을 위한 소프트웨어 개발 환경 구현

최 영 한† · 김 성 운††

요 약

본 논문은 SDL로 쓰여진 통신 프로토콜을 검정하고 명세하기 위한 통신 소프트웨어 개발 환경 구현에 대해 기술한다. 이 환경은 상업용 소프트웨어 개발을 위한 Concerto라 불리는 소프트웨어 팩토리를 이용해 구현되었다. Concerto는 실시간 시스템 또는 통신 프로토콜 분야의 응용 소프트웨어 개발을 위해 설계된 소프트웨어 팩토리로 해당 소프트웨어 개발 및 구현 자동화까지 여러 가지 도구들을 제공한다. 이러한 과정들은 여러 가지 형태의 formalism 즉 C언어, C++, structured 도구화 기법 및 HOOD(Hierachical Object Oriented Design) 방법들의 손쉬운 통합들을 통해 이루어지는데, 본 논문에서는 통신 프로토콜 소프트웨어 개발 과정에서 필요한 도구들인 SDL(System Description Language) 명세(specification)를 위한 그래픽 편집기 구현과 SDL 명세에 대한 동적 분석기(dynamic analysis tool) 등의 구현에 대해 설명한다. 먼저 구현된 환경에 대한 주요한 기술들을 설명하고, 실제 통신 프로토콜에 적용하기 위해 Inres 프로토콜을 예를 들어 설명하였다. 또, Concerto의 하이퍼텍스트 메커니즘이 해당 프로토콜과 상응되는 서비스 사이의 논리적인 링크를 어떻게 생성하는지에 대한 방법을 기술하였다.

A study on implementation of software development environment for SDL

Young Han Choe† · Sung Un Kim††

ABSTRACT

This paper presents a programming environment for the edition and verification of the specification language SDL. It is implemented in the Concerto software factory, which has been designed as a support for the development of real size software. Concerto is a software factory designed for application development in the fields of computer communication protocols or real time system. It offers various kinds of tools to produce software, organize this production and automate it. Concerto allows a smooth integration of various formalism such as the C and C++ languages, structured documentation, HOOD methodology(Hierarchical Object Oriented Design). In this paper, as an important tools in the process of communication protocol software development, we describe an implementations of SDL graphical editor tool and a tools for the dynamic analysis of SDL specifications. We first describe the principles of this environment, then we illustrate its application by means of a simple example, the Inres protocol. Moreover, we study how the hypertext mechanism of Concerto enables to create logical links between a protocol and its associated service.

† 정 회 원:한국전자통신연구소 선임연구원
†† 정 회 원:부경대학교 정보통신공학과 전임강사
논문접수:1996년 1월 19일, 심사완료:1996년 7월 18일

1. Introduction

It is presently well known that the progress of computer technology is impeded by the gap between the durations of hardware and software development phases.

Whereas hardware can be obtained with reduced cost and size as well as improved speed, the development of software is still a slow and largely disorganized process, very much dependent on isolated individual efforts. The realization of software applications becomes more and more difficult, when the size of the application surpasses a certain level, thus obstructing the evolution of software technologies.

Among the solutions proposed to solve this problem, the approach presented by software engineering environments seems the most promising. A software engineering factory attempts to combine the methods supported by software tools with an organization based on team work.

The methods and their associated tools aim at improving the productivity as well as the quality of the work of software developers, by introducing some automatisms and procedures in the organization of tasks.

The use of standard formal description techniques for communication protocols contributes to this effort. In particular, they propose rigorous formalisms for software description with controlled maintenance and evolution, presenting stable versions used by a very large community of users, and which facilitate the communication between designers and users.

In order to illustrate these aspects of software development, we present a software engineering environment, Concerto [1], and an edition and verification tool for the SDL language based on this software factory. It must be mentioned that the SDL environment that we will describe henceforth is an industrial product based on an existing SDL prototype developed at the Centre National d'Etudes des Télécommunications (CNET, France Télécom) [2]. The main features which differentiate between one another is the graphic editor, which has been rewritten on the X-Window System, and the development of a specific archivist for SDL. the most important differences will be described in detail in the section concerning the SDL environment. An example will then be given, the Inres protocol [3], to illustrate the application of this tool.

This paper is organized as follows. First of all, section 2 gives an outline of the software environment Concerto. Section 3 presents the SDL tool itself, then section 4 illustrates the application of this tool on an example, the Inres protocol. Finally, section 5 gives the conclusions and orientations for future work.

2. An outline of the Concerto software factory

Concerto is a software engineering factory designed for the development of technical and real-time applications [1][4]. It is composed of an integration platform suitable for receiving new tools, as well as existing tools used in the different phases of the software life cycle.

2.1 Architecture

Concerto attempts to bring about a high level of integration and flexibility, by implementing the following elements: a general model of software tools, a multi-archive object manager, an efficient communication mechanism between all the tools and an exhaustive parameterized definition of tools.

A software tool can be seen as a program that creates or modifies objects, usually in an interactive manner. The objects manipulated by the environment can be either produced objects (e.g. programs and documentation), or objects representing the state of these objects (e.g. planning, operating documents). The general model of software tools, as specified in Concerto, consists of a functional decomposition which we can classify as dialogs (with the user), mem-

ory manipulation, read/write operations (loading or arranging created or modified objects), and specific functions which constitute the body of the tool. To connect the above-mentioned functions to the memory containing the objects, three processors are defined: a dialog processor with all the graphic and textual interface functions, a structure processor with functions that manipulates the data in the memory, and an archive processor that supplies functions for transfers between the working memory and secondary memory as well as the management of links between objects. This general tool model enables faster and less expensive development of homogeneous, independent and portable tools in Concerto. This architecture facilitates the realization as well as the integration of software tools in the factory.

Due to the diversity in the size of the objects used in an environment and their organization, we need different object management systems. In some cases, we can use directly the file management system already present in the Operating System (UNIX or VMS); but for more complex configuration, we need a more sophisticated system such as a relational data base or an object management system. Sometimes we also need to incorporate a version management function into the system. Keeping in mind these various needs, Concerto is designed to allow the access to several different archive system at the same time. In order to ensure a correct integration of these different system and also to create links between archive objects, the environment offers a hypertext management link mechanism between heterogeneous archives.

For efficiency and flexibility, the tools integrated in the Concerto environment contain formal descriptions allowing parameterization. A tool includes a description of object construction rules, from which we can generate the object analyzer and syntactic editors. It also contains a description of object representation rules in graphic or textual forms, from which we can create the object presentor, a description of the tool's commands used by the Concerto command interpreter, and finally a precise description of the help messages. The advantages of such descriptions is that the cost of tool development is reduced and maintenance/upgrade is easier. It also allows the construction of generic tools such as a generic editor that can be used for any language present in the software factory (ADA, D, LISP, FORTRAN...), as well as for documentation tools.

2.2 Concerto components

From a user point of view, Concerto appears as a set of components, which can deal with any formalism by means of a common interface. The Concerto platform, which ensures that the external visibility of all tools be uniform, is composed of three software components. The '*Dialog Processor*' provides the user interface. The '*Structure Processor*', also called VTP (Virtual Tree Processor), allows the structured objects to be handled in memory. Finally, the '*Archive Processor*' manages the organization of the objects on disks.

The generic '*Editor*' tool offers a set of commands used to handle both structured and textual objects. It allows structural editing of any language whose grammar has been, previously, formally described within the factory. The commands that are most often used during structural edition are '*develop*' (to develop a complex structure) and '*text edit*' (to develop a terminal structure).

The generic '*Archivist*' tool handles the archive elements containing specifications or programs. As previously mentioned, these archive elements can be simple Unix or VMS files, but they can be also the structured archives such as database elements. This tool offers a common interface to access any archive system. In addition, it manages the hypertext links that can be created between different archive elements.

3. The SDL environment

In this chapter, we first describe the SDL subset

used by this tool. Then we present the different components of this environment, namely the '*structural editor*', the specific '*archive system*' and the '*static semantic checker*'.

3.1 The SDL subset

The SDL tool does not use the SDL standard as defined in the CCITT documents [5], but a subset of this language defined by a French group of industrial and adminstration partners. The main criteria underlying the definition of this subset were the elimination of redundancy from the Z.100 definition, the elimination of concepts that are not of a purely specification nature, and the simplification of complex concepts. The precise description of the SDL subset can be found in [2] and [6].

3.2 The structural editor

The main facility offered by the Concerto software factory is the possibility to create textual structural editors for any language with generic commands. In the case of SDL, which possesses two different syntactic forms, a graphic one(SDL/GR) and a textual one (SDL/PR), specific SDL tool, with its own menu, was necessary to give a user interface with specific graphic commands. We first describe the combined use of graphic and textual editing, then we present the main commands of the SDL tool. Finally we shall give some comments on how to use the environment in an efficient way, by means of the multi-windowing capabilities of the tool.

3.2.1 A graphical and textual editor

All the commands that should have an impact on graphic views are executed by means of the SDL tool, while all other commands are obtained by using the classical, generic '*Editor*' tool. For example, if one wishes to create a new process inside a block, '*create element*' command of the SDL menu will be used. The parameters required for this command, such as the name of the new process, will be typed in by the user in a choice block opened by Concerto. Once the process has been created, with the corresponding modifications in the graphic views, it will be necessary to detail its own textual definitions, such as variable declarations, timer declarations, etc. These declarations, which are purely textual and have no impact at all on the graphic views, will be typed in by means of the usual commands of the '*Editor*' menu, namely '*develop*' (to develop a complex structure) and '*text edit*' (to develop a terminal structure).

Creation commands

The command which is most often used in the SDL tool is undoubtedly the command '*create element*'. Indeed, this command allows to create a new element, at a given position in the SDL specification. Note that we deal with logical positions, and not graphical positions. For example, we can create a process after a given existing process, but graphical coordinates in the graphic views are taken into account automatically by the editor. One can create an element in, before or after any existing element. The feasibility of the creation at the given position is then tested, and the user is asked for the parameters necessary to characterize the required element. In particular, if several different types are possible at a given position, he will have to choose one of them. For example, inside a system, one can create either a terminal block, a block substructure or a block reference. All parameters typed in by the user are then checked lexically according to the SDL standard before the element is actually created. Hence, the editor enables creations only if they are correct lexically and syntactically.

There also exists an alternative to the former command, which is the command '*create branch*'. This command allows to create a new branch, at a given position is the SDL specification. This branch can be either a stimulus branch(input, enabling condition or save, after a state) or a decision branch(answer branch or else branch, after a decision). Once again, we deal with logical positions, not with graphical ones. One can create a branch in, before or after an

element or a branch, insofar as the SDL syntax is respected.

As for communication between several SDL entities, one should use the command '*create link*'. This command allows to create a link, i.e. a channel or a signal-route, between two entities. The link may be unidirectional or bidirectional. The graphic position of the link is automatically computed by the editor, without any manual intervention of the user.

A particular command is used to create SDL connections. The command '*connect*' allows to create a connection between one channel external to a given block, and one or several sublinks(channels or signal-routes) internal to that block. For ease of use, external channels appear in the periphery of the block in graphic views, so that the user can select in the same window both the external channel and the internal sub-links.

Finally, the command '*create model*' has been added to the SDL menu in order to allow the user to create a new, empty SDL specification.

Destruction commands

Most of the time, destructions of SDL entities in a SDL specification are obtained by use of the command '*delete*'. This command allows to delete one or several elements or branches, if this is permitted by the syntax. Some elements cannot be deleted, such as the start element of a process which is compulsory. Remember that the commands of the SDL tool can be used only for actions which have some impact on graphic views. Consequently, if one wishes to delete a purely textual element of information, such as a variable declaration, he will have to use the '*delete*' command of the '*Editor*' tool, not the '*delete*' command of the SDL tool.

As for the command '*disconnect*', it allows to delete a connection between one channel external to a given block, and one sub-link(channel or signal-route) internal to that block.

Modification commands

The command '*modify*' allows to modify one element or branch. The former parameters characteristic of the element(type, name...) are presented to the user in a choice block. He can then type in the new parameters required, or even change the type of the element. The validity of each modification is, of course, checked according to the lexical and syntactic rules of the language.

Another command, '*reverse*', is used in very special cases. It allows to reverse the direction of the graphic symbols for input an output elements. This has no influence at all on textual views, but in some cases it seems more convenient to reverse the graphic symbols.

Moving commands

When one wishes to move any element of a SDL specification, the user can choose between one of the following commands: '*insert*', '*move*', and '*exchange*'. For the first two commands, the user must indicate the target position for the moved object.

Other commands

The command '*show*' allows to open a new, graphic or textual, view centered on any element of a SDL specification. This is the key command for the sue of the multi-windowing facilities of the environment. Moreover, it will also be useful to open an error view, graphic or textual, to display the errors possibly detected by the static analyzer.

The command '*print*' allows to print an element of a SDL specification. A special section is devoted to printing facilities in the environment.

As for the command '*check*', it allows to check the static semantics of an element. This facility will also be described in the following section.

3.2.2 Main commands of the editor

The different commands of the SDL tool, which are presented to the user in the SDL menu, can be classified into several classes: creation commands, destruction commands, modification commands, moving commands... Most of these commands are generic, in the sense that the same command can be used to deal with elements of different types, both in the architec-

ture or in the automation part of the specification. Moreover, it is important to note that the arguments of those commands can be selected either in a graphic or in a textual view. Finally, each of these commands can be undone by means of the generic '*undo*' command integrated in Concerto.

3.2.3 A multi-window editor

One of the main advantages of Concerto as compared to many other software factories is the multi-windowing facilities that are offered. These facilities are also present in the graphic tools. In the SDL environment, it is possible to have any number of graphic and textual views of the same specification, each one centered on a specific park of the SDL model. This facility can be used in the following way: if you open two graphic views, one centered on a smaller part of this state in a larger window, then it will be possible to work with quite a great level of detail on the local part of the state that is currently being modified, while still having a global view of the whole state. Of course, one textual view (at least) will also be necessary to type in purely textual information.

3.2.4 Printing

The SDL environment enables to print any element of a specification in three possible ways. First, you can print the textual description of the element, that is to say the content of any textual view opened on that element. Second, you can print the graphical view of the element alone. Finally, you can recursively print the graphical diagrams corresponding to the element itself and all its hierarchy. This last solution is used, for example, to print a whole specification.

Note that printing algorithms are quite different if one wishes to print a structure or an automaton element. In the case of structure elements, diagrams are reduced in order to fall into an A4 sheet of paper. In the case of automaton elements, the dimensions of graphical symbols are normalized by CCITT, and such a reduction technique cannot be applied. Hence, truncation algorithms are used to print a single state on several pages, and cross-reference graphical symbols are used to link these different sheets between one another.

3.3 Archive system

When no specific archive system exists for a given formalism, one simply stores the corresponding programs into Unix files. This is of course possible for SDL specifications. They can be stored into Unix files, or opened from Unix files, so as to permit exchanges with other SDL tools. Though, when one is working inside Concerto, it is much more convenient to use the specific SDL archive system, where specifications are stored in an internal form. Not only does this form occupy less space on the disk, but altogether archiving facilities are much more practical, as we shall show below.

3.3.1 Archive elements

Archive elements are the smallest elements of programs that can be treated by an archivist. In the case of SDL specifications, these elements include all the architecture elements up to the states included. The consequence is that one will be able to open and store directly a process or a state, without having to open or store the whole specification. This is achieved by means of the SDL catalog.

3.3.2 The SDL catalog

An archive system is mainly based on the notion of catalog. In the case of the SDL environment, the user specifies a certain number of SDL bases, which are Unix path-names indicating where specifications will be stored and manipulated in internal form. The SDL catalog then presents the list of all specifications present in each SDL base. These specifications will be opened and stored just by selecting them in this catalog view. If one wishes to open, not a whole specification, but a smaller element of a specification, such as a state, another archive view than the catalog

view must be used, called the observation view. This view gives more information than the catalog view, in the sense that it presents, for each specification, the hierarchy of blocks, processes and states that compose the specification. Any archive element can then be opened and stored just by selecting it in this observation view. Another facility offered by these two types of views is to indicate, by an asterisk, the elements that have been modified by the user since the last save.

3.3.3 Main commands of the archivist

We present below the main commands that are used for archiving purposes in the SDL environment. It is important to note that these commands are generic. In fact, they appear in the '*Archivist*' menu, which is a generic tool in the Concerto software factory. The same commands can consequently be used for any kind of formalism.

The command '*catalog*' is used to open the catalog view of the specific SDL archive system, which presents the list of all the available specifications.

The command '*observe*' enables to open an observation view on any SDL specification.

Another command which is very often used is the command '*open*'. It enables to open an archive element, which can be either a whole specification or just a part of it. Note that the same command can be used to open a SDL specification from a Unix file.

The command '*store*' is needed to save an archive element. The catalog and observation views are updated according to the modifications which are being stored on disk. Once again, the same command can be used to store a SDL specification into a Unix file.

The command '*close*' is used to close all the views opened on the selected element, either graphic or textual. If the element has been modified since the last save, the user will be asked whether or not to save it.

Finally, the role of the command '*reference*' is to implement the hypertext mechanism that we present is detail in the next paragraph.

3.3.4 Hypertext mechanism

The hypertext mechanism is another important feature which forms the basis of Concerto. In fact, the design of archive systems in a generic way enables to reference, in any structure, any archive element. Let us give two examples in the case of the SDL environment, related to the usual software life cycle. With the same generic command, namely the command '*reference*' of the '*Archivist*' tool, it will be possible both:

- to reference a SDL state in a paragraph of an informal text of the Concerto's documentation environment, describing the specification,
- to reference a C or C^{++} program in a SDL state, whose behaviour can be represented by this program after the code generation phase.

Hence, the hypertext mechanism appears as a function essential to carry out the traceability studies necessary throughout the life cycle of a system or a software.

3.4 Static semantic checker

This tool enables to check the static semantics of any element of a SDL specification. According to the SDL standard, it implements an important number of different types of check such as an interactive simulator, an exhaustive semantic checker based on the a global state-transition graph, and a temporal logic checker working on this graph. Moreover, a gravity level is associated to each static error detected by the tool, ranging from simple warnings to fatal errors. Once a specification or a part of it has been checked, the corresponding error messages can be visualized in a user friendly fashion. The same '*show*' command of the SDL tool can be used to open graphic or textual error views on any element. In a graphic error windows, they present only the substructures of the specification that are erroneous, preceded by the corresponding error message and its gravity. SDL specifications can then be corrected directly in these views. Finally, these exist a mechanism to mark the errors that are considered by the user as probably corrected, before the next semantic check is performed.

4. Application of the SDL environment to the Inres protocol

We have used the SDL environment for the specification and the verification of both the Inres protocol and the associated service. This protocol does not correspond to any international standard, but it presents many software aspects specific to communication protocols. This is the reason why we have decided to apply our tools on this system. We first describe briefly the Inres protocol, then we present our work with the SDL environment. Finally, we shall show how we have used the hypertext mechanism of Concerto, in order to establish dynamic links between the protocol and the associated service.

4.1 The Inres protocol

Though it does not correspond to any real network, the Inres (Initiator-Responder) protocol [3] is quite interesting, since it presents many basic OSI concepts and mechanisms, such as connection, disconnection, sequence numbers, acknowledgements and retransmission on time-out. It is in fact a simplified version of the Abracadabra protocol described in [7]. We first give an informal description of the protocol and its associated service, then we shall present the SDL specification of the protocol and its associated service, then we shall present the SDL specification of the Inres service and its analysis by means of the SDL environment.

4.1.1 Service outline

The Inres protocol is connection oriented and asymmetric. Hence we represent the communication between an initiator entity, which establishes the connections and sends data, and a responder entity, which both accepts and releases connections, and receives data.

Before any data exchange between the two entities is accomplished, a connection muse be established through the classical exchange of four service primitives, denoted ICONreq (Inres connection request), ICONind (INres connection indication), ICONresp(Inres connection response) and ICONconf(Inres connection confirmation). Once the connection has been successfully established, the initiator entity can transmit data by means of the IDATreq(Inres data request) service primitive, which will be delivered at the responder entity as IDATind(Inres data indication). After completion of the data transmission, the connection will be terminated by the responder, which issues IDISreq (Inres disconnection request), delivered at the initiator as IDISind(Inres disconnection indication). A number oproblems may arise, such as rejection of a connection request by the responder, or erroneous transmission of the various data involved in the service. Most of these problems cause a IDISind to be received by the initiator entity.

4.1.2 Protocol outline

To achieve the reliable, connection-oriented service described earlier, the Inres protocol uses the service offered by a communication medium between the two entities. The service of this medium is symmetrical and unreliable, in the sense that it may lose data. Let us now describe briefly the protocol itself.

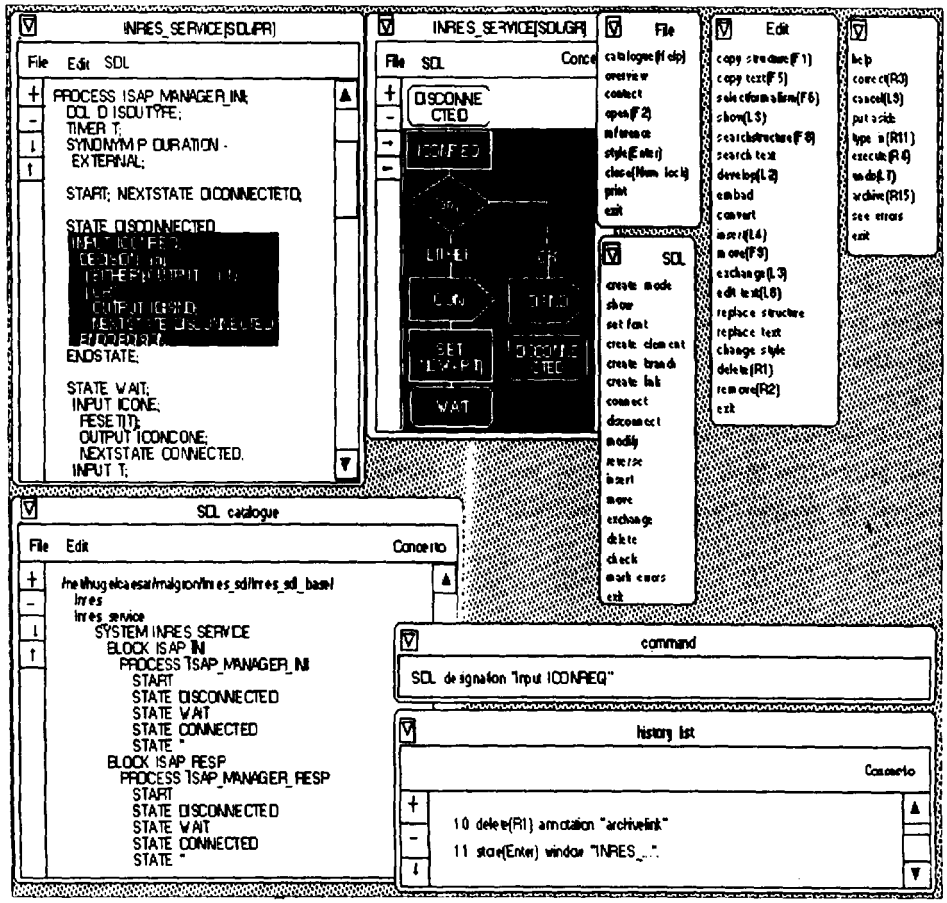
The communication between the two entities takes place in three successive phases: connection establishment, data transmission and disconnection. During the connection establishment phase, two protocol data units are used, namely CR (connection request) and CC(connection confirmation). Thereafter, the data transfer phase uses two other protocol data units: DT(data transfer), which contains both the service data unit itself and a (one-bit) sequence number, and AK(acknowledgement), which contains only the sequence number of the message it acknowledges. Here, the concept of sequence number is analogous to that of the control bit in the well known alternating bit protocol: the initiator entity which has just sent a DT with a given sequence number waits for the corresponding AK, before it is allowed to send the next

DT with the next sequence number. If it receives an AK with a wrong sequence number, it simply retransmits the last DT. Finally, a DR(disconnection request) protocol data unit is used during the disconnection phase. Throughout the Inres protocol, retransmission over time-outs are used to take into account data losses inherent to the medium service.

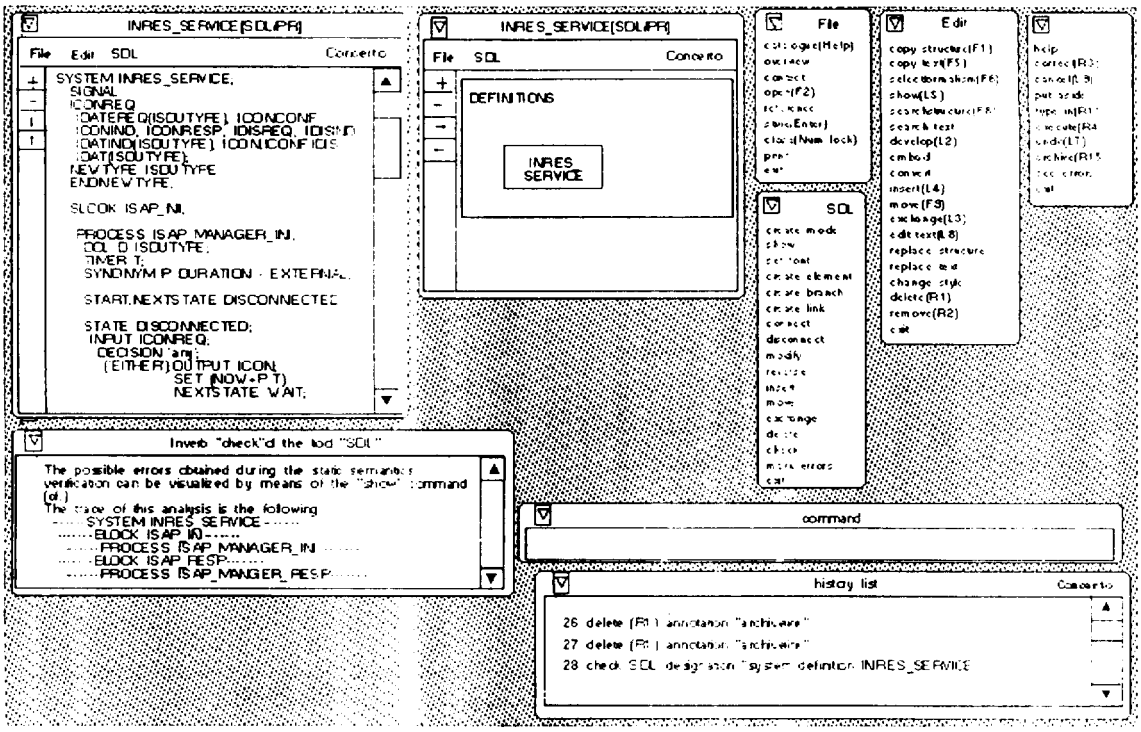
4.2 SDL specification and analysis

We have modeled the Inres protocol and service in SDL, by means of the graphical editor described earlier. Both specifications have been stored in the SDL archive system. We then verified the static semantics of both models with the SDL tool.

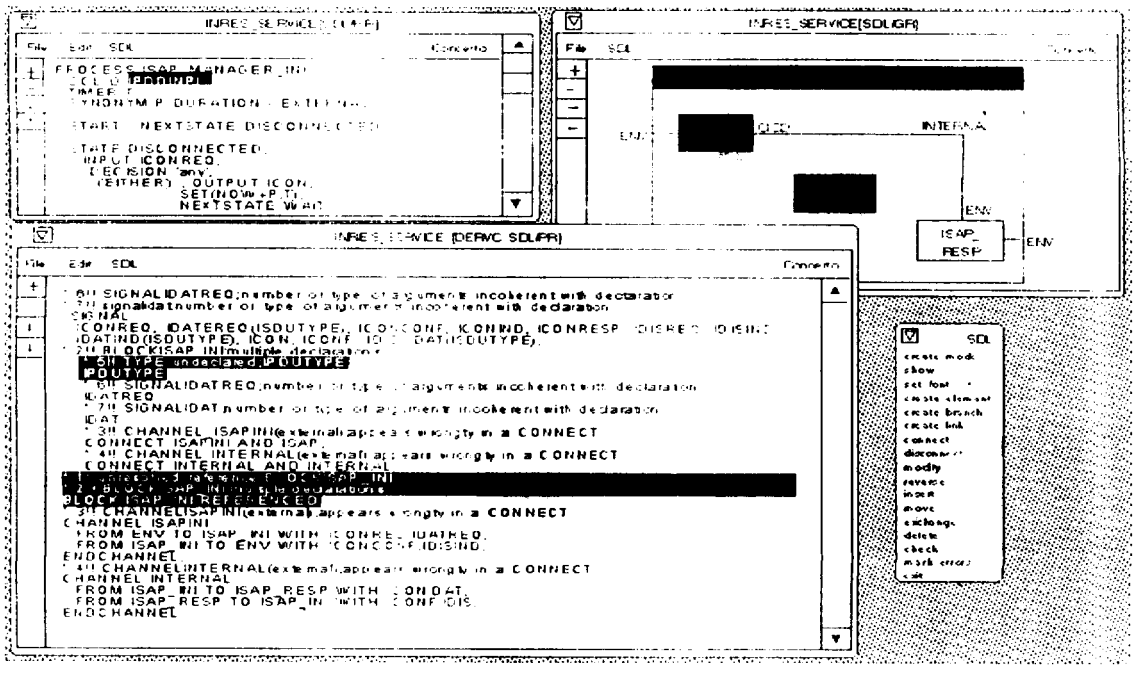
We shall present several screendumps, related to the different views of the SDL system in a Concerto session. Thus, figure 1 shows at the same time a view of the SDL catalog, as well as a graphic and a textual view of the same specification. It is important to point out that an element selected in one of the two windows opened on the specification is highlighted in both. Figure 2 shows that the verification of the static semantics of the Inres service does not detect any error. Finally, in figure 3, we have intentionally added several static semantics errors in our specification, so as to show how the SDL tool displays them.



(Fig. 1) The SDL environment of Concerto



(Fig. 2) Verification of the static semantics of the Inres service



(Fig. 3) Visualization of static semantics errors

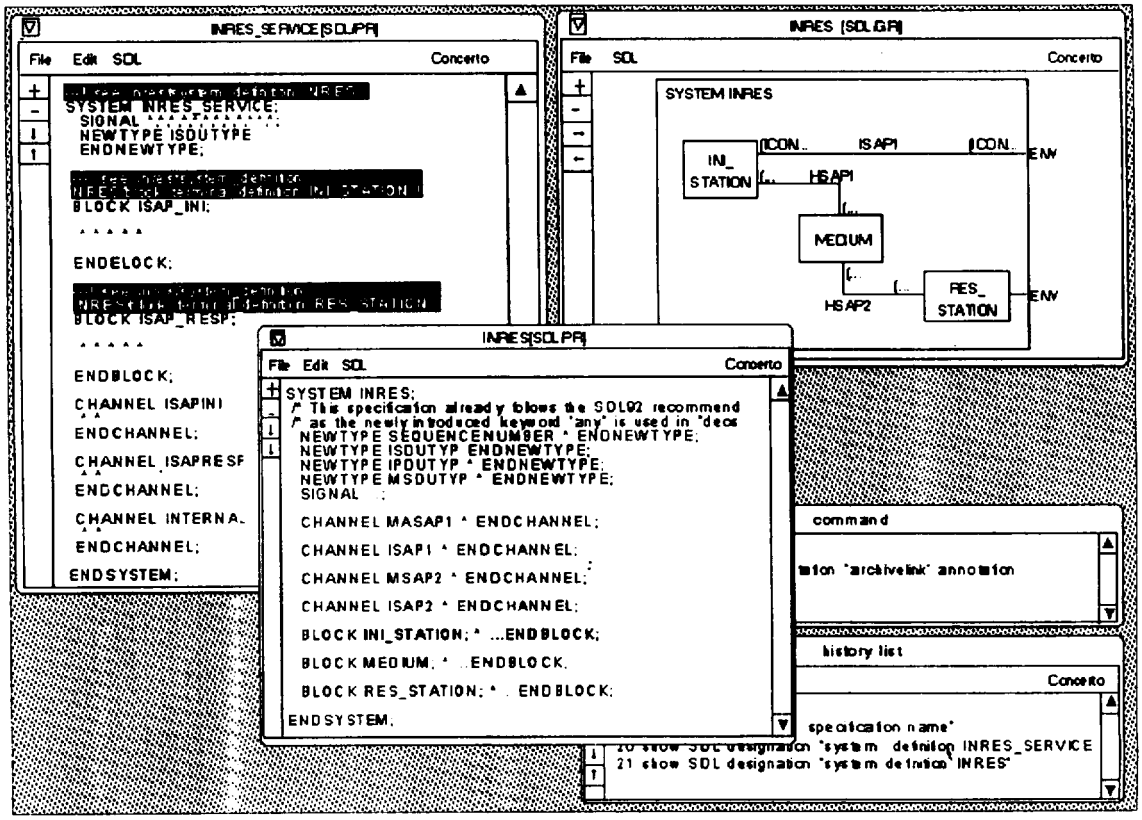
4.3 Relationships between protocol and service

The last phase of our work is the creation of links between the specifications of the Inres protocol and the Inres service. We have used for this purpose the hypertext mechanism present in Concerto. The reader can observe, in figure 4, that archive links have been created in the service specification, pointing towards the protocol specification. These links concern, not only the whole specification, but also some of their constitutive elements, such as blocks or processes. These hypertext links can then be used to find out the elements of the protocol specification which implement a given element of the service specification, and conversely.

5. Conclusions

This work points out that software engineering factories, such as Concerto, seem to be necessary frameworks for the development of powerful edition and verification tools. In this paper, we have presented an software development environment for SDL language, and illustrated its application on a communication protocol. One of the main advantages of this environment as compared to many other software environments is the offered multi windowing facilities. And also, the hypertext mechanism is another important feature which forms the basis of this environment.

According to a given SDL specifications, we implemented some different types of dynamic checking such as an interactive simulator, an exhaustive semantic checker based on the construction of a global state



(Fig. 4) Use of hypertext links in the SDL environment of Concerto

transition graph, and a temporal logic checker working on this graph.

Moreover, we have created several links between different parts of the protocol and service specifications. These links, which are based on the hypertext mechanism of Concerto, will be used in a future work to facilitate the conformance verification between protocols and associated services[8][9].

REFERENCES

[1] CONCERTO, "l'atelier de génie logiciel, présentation générale", Sema Group, October 1991.

[2] J. Camacho, C. Langlois and E. Paul, ELVIS, "an integrated SDL environment", Proceedings of the 4th SDL Forum, Lisbon, Portugal, 9-13 October 1989.

[3] D. Hogrefe, "OSI formal specification case study: the Inres protocol and service", Universität Bern, Institut für Informatik und Angewandte Mathematik, May 1992.

[4] A. Conchon, "Projet Concerto: le poste de travail", L'Echo des Recherches, N°119/120, France, 1985.

[5] SDL, "Functional and Specification Language", Recommendation Z. 100, CCITT, Geneva, 1989.

[6] C. Langlois, M. Martin, A. Rouger and E. Paul, "Définition du sous-ensemble LDS88 proposé par le CNET", Note Technique NT/PAA/CLC/LSC/2236, CNET Paris A, July 1988.

[7] "Guidelines for the application of Estelle, LOTOS and SDL", ISO TC97/SC21, ISO Technical Report 10167, 1990.

[8] A. R. Cavalli, S. U. Kim and P. Maigron, "Automated protocol conformance test generation based on formal methods for LOTOS specifications", 5th International Workshop on Protocol Test Systems, Montréal, Québec, Canada, September 1992.

[9] H. Fouchal, A. cavalli and Sung Un Kim "An Integrated Tool for LOTOS development", IEEE, In'tl Conference on Computer and Communication, Seoul, 1995.

최 영 한

1981년 경북대학교 전자공학과(공학사)

1992년 충남대학원 전산학과(이학석사)

1987년~1989년 AT&T Bell Labs. 방문 연구원

1993년~현재 ITU-T SG7 Q.17 Editor

1993년~1994년 ETSI MTS 회원

1982년~현재 한국전자통신연구소 프로토콜 기술연구실 NTB과제 책임자.

관심분야: 프로토콜시험, 초고속정보통신, 정보통신 표준화, 컴퓨터네트워크

김 성 운

1982년 경북대학교 전자공학과(공학사)

1990년 프랑스 국립파리 7 대학교 정보공학과(공학석사)

1993년 프랑스 국립파리 7 대학교 정보공학과(공학박사)

1982년~1985년 한국전자통신연구소 데이터통신연구실(연구원)

1986년~1995년 한국통신 연구개발원(선임연구원, 실장)

1990년~1993년 프랑스 전기통신기술연구소(초빙연구원)

1995년~현재 부경대학교 정보통신공학과(교수)

관심분야: 프로토콜 엔지니어링, 데이터 통신 통신프로토콜시험, 컴퓨터네트워크