Bell Canada's Generic Network Element Control Technology

Authors:

Bernard Stepien, **Protocol Standards Corporation(PSC)**

Carl Ward, **Bell Canada**

Date:

April 4th 1991

Abstract:

The NEC software development project is put into context in its operational environment at Bell Canada. The NEC controls many network control functions of Bell's Special Services Digital Network. Special purpose languages have been designed for application development in the SSDN. At the heart of the software lies the Inferencing Connection Engine (ICE). Its overall architecture is described along with the linguistic mechanisms which drive it.

Introduction and Overview of the NEC

Bell Canada is implementing a control architecture to centralize the operations, administration and maintenance of a Special Services Digital Network (SSDN) . The control architecture is being achieved through the creation and deployment of the Network Element Controller (NEC) which centralizes access to Digital Cross Connects (DCC) and Intelligent Channel Banks (ICB). The NEC is a software application which defines and uses advanced concepts of computer languages. It has its own proprietary languages (ICE, FDL) developed by Kylain Technologies Inc. It has its own set of compilers for these languages. It currently runs on a VAX platform and uses Ingres as it data base server. The NEC has three main objectives:

1) secure, partitioned access to Bell Canada's DCC and ICB.
2) consistent interface for all accesses to SSDN Network Elements (NEs) including provisioning and testing
3) protocol mediation between NEs (to/from: PDS, MML, TL1, ASN.1)

The NEC concept enables the representation, (using an abstract model) of any network element machine. The model is defined through the use of individual transactions in the area of:

• provisioning
• testing
• surveillance
• configuration
• administration

The NEC is a centralizing system. The notion of centralization is more about the centralizing of network information rather than centralizing the provisioning (or testing...) activity. Intervention on the network is performed by geographically scattered independent users through user friendly interfaces that are connected to a central NEC system.

The variety of equipment used to perform the DCC and ICB functions naturally generates two basic requirements for such a software environment:

- flexibility in dealing with multi version controlling software of existing NEs
- adaptability to future generations of NE equipment and services.

There are four main components to the NEC system:

- a user interface
- a communications manager
- an operating system
- a database system

The basic flow of data consists in

OUTGOING: (to the elements)
transactions (commands) flowing:

a) from the user interfaces to the network elements in order to perform some provisioning action and

b) to the database to record that action for audit and administrative purposes, and

INCOMING: (from the elements)
all the messages that will come back from the network elements

a) responses to the user interface
b) acknowledgement of command acceptance,
c) echoing in case of full duplexes
d) alarms

all this through the communications manager component..

An Example

If one would want to establish a cross-connect between two in-service digroups, a cross-connection provisioning transaction of the following form would be issued (after filling in the information on a user interface screen(s)):

connect DNX-TOR23 digroup *25* channel *6*
to digroup *73* channel *5*
signal bits *10010011* pcm bits *GF*
disconnect signal *01011101* pcm *60* <cr> <lf>

This format of command is not understandable by the target DNX (the DNX is a DCC manufactured by Northern Telecom), and has to be converted. In the conversion process, one has to know first of all what to convert (identify the class of command), and then find the equivalent format for the same class of command in the target machine and perform the appropriate conversion.

The above command will be ultimately be converted to the following Man Machine Language command:

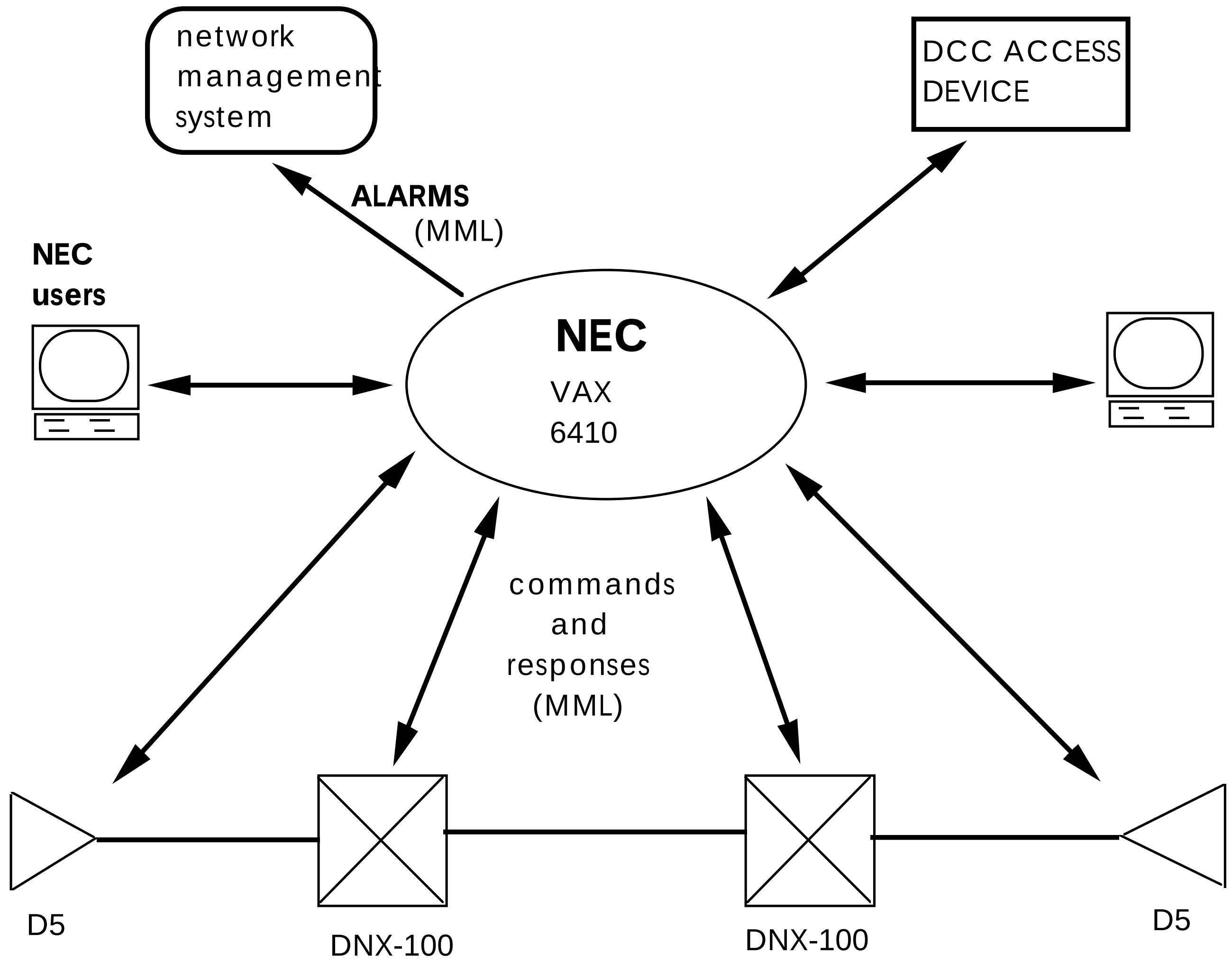
CON-TW-DS0:25-6,73-5:b'10010011,GF,b'01011101

Once this command is processed by the target DNX, the DNX will respond with the appropriate responses such as 'ACCEPTED' or 'REFUSED' depending of the actual state of the DNX when receiving this command.

This however is not the only function of the NEC system. So far, the above example has only achieved a remote access function. For system administration, one needs to record the new state of the DNX, if the cross-connection has succeeded, or to undo successful connections in a circuit where some connection attempts have failed.

This is achieved by storing all successful provisioning results in a database. This in turn generates a family of database administration functions. Since manual operation of a DNX still remains possible in case of failure of the NEC and for emergencies, a misalignment between the true state of a DNX and its database image may occur. Consequently, from time to time, one has to realign the stored information. This is an on-going background task of the NEC.

Currently the system has been implemented on a VAX 6420 with a back-up 6420 on standby for switchover in case of failure. It controls NT's DNX-100 Digital Cross Connects and AT&T's D5 Intelligent Channel Banks for the T1 network from either user terminals or another existing DCC access device. (figure 1). Future releases will control NT's DE4-E-Smart ICB and future NT FibreWorld products (integrated DCC and ICB).



where MML stands for Man Machine Language

figure 1

We shall now discuss the various features of the NEC and its technology.

Application Specification Languages

There are four description languages in the NEC:

• Format Description Language (FDL)
• Protocol Description Language (PDL)
• Language for User Interface (LUI)
• Data Definition Description Language (DDDL)

These languages are described using an augmented Backus-Naur form (ABNF) that provides both the syntax and the semantic constructs required for execution. The semantic construct are based on reusable primitives that can be assembled into any appropriate combinations to perform a function described by the syntactic element. These languages are compiled by the Language Specification Mechanism compiler to produce a compiled version of the language specification. The compiled version is then used along with an application specification source code by a compiler-compiler to produce a compiled version of the application. (see figure 2)

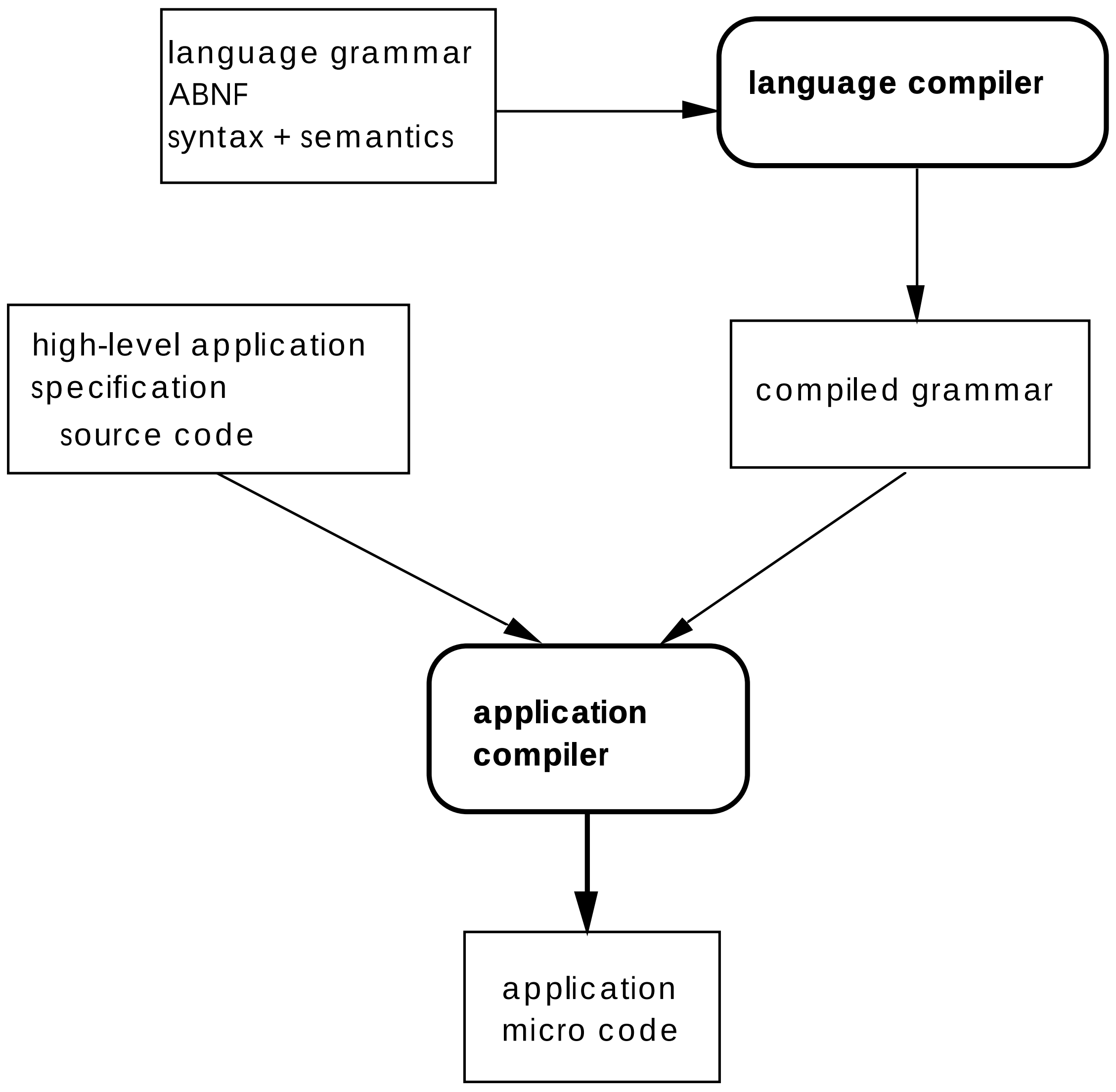

language grammar
ABNF
syntax + semantics
language compiler
high-level application specification source code
compiled grammar
application compiler
application micro code

figure 2

ABNF Example

The following example shows the ABNF for a protocol specification in the Protocol Definition Language :

```
:protocol:
(
   "protocol"                              [set][lit][break_address][MAXINT]
                                                   [reset][self_describing]
                                                   [reset][transport_descriptor]
   *protocol_name*
                                                   [set][val]
                                                   [program_name][*protocol_name*]

   (|  "using" "transport" "descriptor"
       *descriptor_name*           [set][val][transport_descriptor]
                                                   [*descriptor_name*]
    |  "self" "describing"         [set][lit][self_describing][TRUE]
    |)

   "["                                             [protocol]
   < (1:0) transaction
   >
   "]"
);
```

On the left hand side we find the usual BNF grammar definition, while on the right hand side we find the semantics associated with each production that will be the resulting token stream upon compilation of an application definition.

This approach has the benefit of saving considerable time on re-programming. Adding a new language element merely consist of adding the necessary few lines of ABNF definition.

The following example shows how one obtains a token stream from a protocol definition using the above ABNF for the protocol production:

specification source:

```
protocol DNX
        using transport descriptor mmltransport
        [

        transaction prov_trans of type con_tw
                using descriptor dnx_con_tw_b
                when exists(Type_connect_b)
        [
        method provision("second_half");
        ]
        ]
```

resulting token stream:

token sequence number	token value	token meaning (primitive)	corresponding specification source
#205	78	set	protocol
#206	79	lit	
#207	58	break_address	
#208	31	2147483647	
#209	80	reset	
#210	35	self_describing	
#211	80	reset	
#212	42	transport_descriptor	
#213	78	set	
#214	81	val	
#215	33	program_name	
#216	0	index of generic names array	"DNX"
#217	78	set	using
#218	81	val	transport
#219	42	transport_descriptor	descriptor
#220	1	index of generic names array	"mmltransport"
#221	82	protocol	
		etc ...	

The above protocol example reveals the use of data descriptors called Format Descriptors (FD).

Format Description Language

A format descriptor is composed of structural information that defines the hierarchy of fields and their lengths but it also gives all the alternative formats a record may have. For example the mmltransport descriptor (Man Machine Language transport) that is used in the above DNX protocol will show the format of the various commands that one can send to a DNX.

```
record mmltransport;                        (length _RECORD_LENGTH;
                                                    format cs_ASCII;)
        [
        field _RECORD_LENGTH;           (context;)
        field _ERROR;                   (context;)

        field clli;                             (length 11;)
        field sequence;                 (length 4;)

        field _command;         (length _RECORD_LENGTH - 15;)
        [
        case _command;
        {   = "DISPL-FWREL";    field Type_querywho;       (length 1;)  }
        {   = "DISPL-DGSTAT";   field Type_queryport;      (length 1;)  }
        {   = "DISPL-TW";       field Type_querycross; (length 1;)  }
        {   = "DISPL";          field Type_query_2;        (length 1;)  }
        {   = "CON-TW";                 field Type_connect;        (length 1;)  }
        {   = "CON-TW";                 field Type_connect_a;      (length 1;)  }
        {   = "CON-TW";                 field Type_connect_b;      (length 1;)  }
        {   = "DISC-TW";                field Type_disconnect;     (length 1;)  }
        {   = "PROV-FAC";       field Type_prov_fac;       (length 1;)  }
        {   = "DISC-FAC";               field Type_disc_fac;       (length 1;)  }
        etc...
```

In this example the **_command** field is depicted through a case statement that gives the alternative ways this field appears. In this case, a given string value representing the MML commands determines the field type.

Protocol Description Language

It is a declarative and procedural language. The basic data being exchanged between the various communicating entities is represented by a transaction which will contain commands to network element or responses from these same network elements. The transaction is consequently the main carrier of information in the NEC system. The transaction is also the basic source of information to drive the inferencing connection engine.

Each protocol has a name that is used to invoke it. It then has two main parts: a transaction definition statement and a method statement. The transaction definition describes how the system can identify this transaction, and upon proper identification trigger a set of actions that are described separately in a method definition block. The format descriptors are used to detect the appropriate transaction. A transaction is always associated with a format descriptor. The **when exists(field_name)** statement allows the identification of the transaction.

In the above example, we see that a transaction is a prov_trans transaction if it contains a field type: Type_connect_b, which is a command that starts with the string "CON-TW" according to the mmltransport format descriptor.

Once the appropriate transaction is identified, we can execute the actions associated with it, in this case the method provision.

A method is a way to further structure a protocol description. It is parametric, and resembles a "C" switch statement.

```
     define method provision ( function )
     [
             case
             [
             arg(function) == "normal"
             [
                     forward transaction prov_trans
                     to :prov_trans:clli
                     window factor 1;
             ]

 ...

             arg(function) == "second_half"
             [
                     forward transaction prov_trans
                     to :prov_trans:clli
                     window factor -1;
                     return;         ]
             ]

     expect transaction acknowledgement
             of type acknowledgement
             within 180 seconds
             with location(transaction acknowledgement)
                     == :prov_trans:clli
     etc...
```

There are various action statements:

- the send statement creates a new transaction and forwards it to the specified location.
- the forward statement forwards an existing transaction to the specified location.
- the expect statement specifies that a transaction of the given type should arrive within a specified period of time.

The Execution Environment

The system is decomposed into modules that are dedicated to specific functionalities. The core of the system is the Inference Connection Engine (ICE) that executes protocols. The execution of a protocol will result in transactions being sent to or from Network Elements, users or other access devices. Some of the functions involved in this process have been separated into these modules:

There are three main modules:

- the system control module, that is a supervisor, starting up or shutting down other modules (PCP).
- the Generalized Driver Framer that handles the communication aspect of the system (GDF).
- the Inference Connection Engine that executes the various protocols.

We have mentioned earlier that this system has been designed to be generic. This aspect has been extended further than merely executing protocols. It is used to further break down the various functionalities of the system. Consequently, there are four copies of the same Inference Connection Engine running in parallel, each of them handling a specific segment of the communication destinations.

Consequently, there is an instance of ICE strictly handling users interfaces, another handling database operations, another handling alarms and finally one handling the network element via the Generalized Driver Framer. The reason for this architecture is of course to take advantage of the asynchronous nature of the various events occuring through the system.

The overall software architecture is depicted in figure 3.

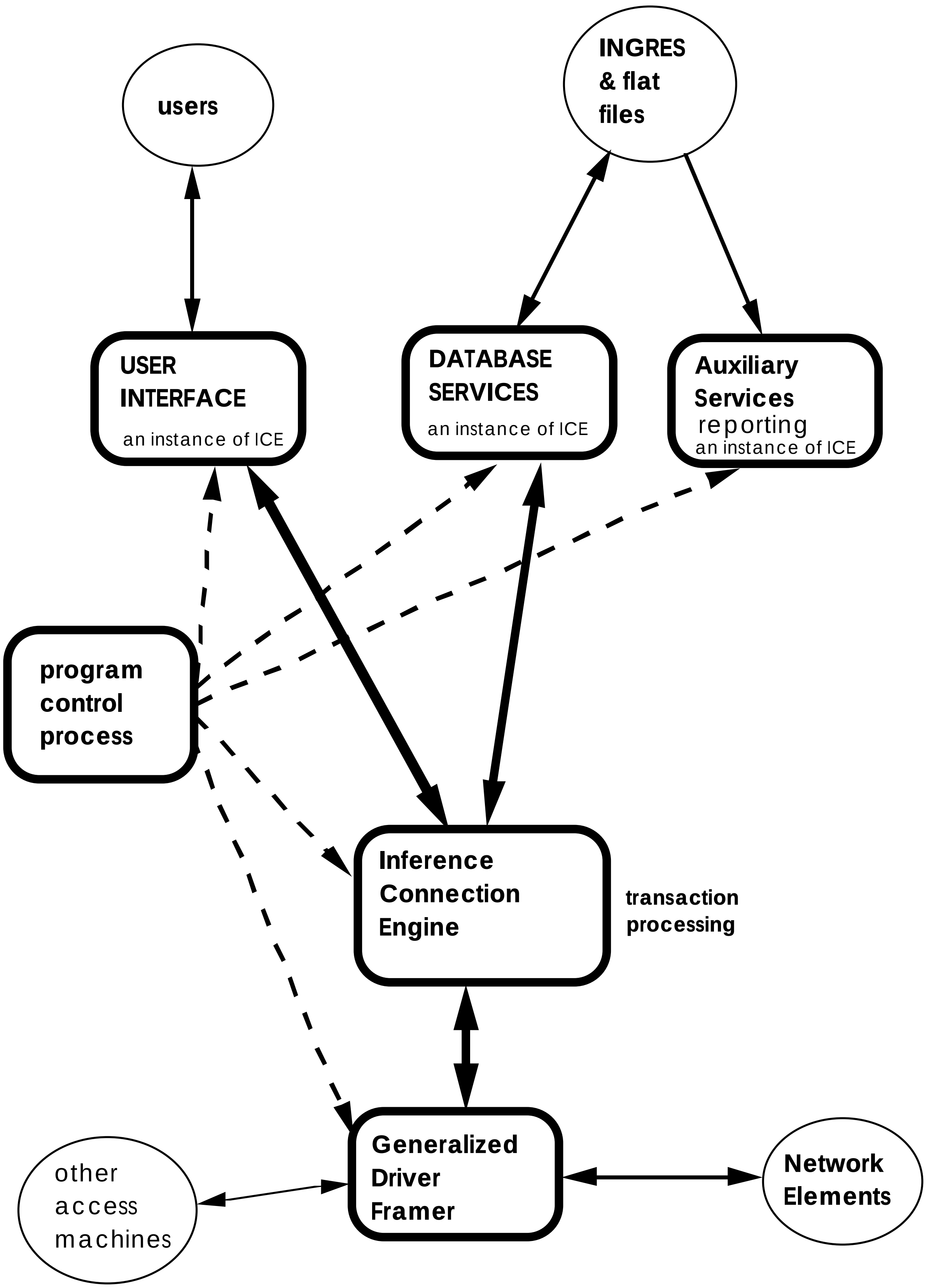

users
INGRES & flat files
USER INTERFACE
an instance of ICE
DATABASE SERVICES
an instance of ICE
Auxiliary Services
reporting
an instance of ICE
program control process
Inference Connection Engine
transaction processing
other access machines
Generalized Driver Framer
Network Elements

figure 3

The Inference Connection Engine

The two main functions of the Inference Connection Engine are to execute protocols and process user interface events. As we have seen above, protocols are specified using the Protocol Description Language and are compiled in executable micro code. The execution of a protocol is performed through processes that are created every time a protocol is used to drive the interaction flow with various Network Elements.

A Process Status Block (PSB) is created for each instance of a protocol execution. The PSB is very similar to the Program Control Block concept of operating systems. It has its owns stacks, registers and program counter to execute a given protocol. A virtual machine is created for each PSB.

The execution of a PSB is dependant of its status

- runnable,
- running,
- holding,
- Scheduled execution is pending,
- cancelled,
- Original transaction exists,
- Expected transaction exists,
- Cancellation of subprocesses required,
- Transmit type transaction,
- Receive type transaction.

Processes are queued, and the ICE system will check this queue to find a runnable process and run it until it enters a new state where it will be holding to expect a response from a communicating party, etc...

The execution of the compiled protocol is performed using an object oriented approach. Each token of the compiled protocols token stream is a primitive that is dispatched to a given C function that handles only one primitive at once.

The selection of the appropriate protocol to execute is determined by the location (network element or user). The normal life cycle of processes consists usually of:

- a process being created when a user logs in to the NEC system,
- this user will eventually trigger the creation of a transaction corresponding to a provision action,
- this transaction will be converted to a new transaction in a format appropriate to the destination,
- a new process to execute the protocol of the destination is created.

This process will receive a response from the location that will generate a response transaction that will be converted back into the user interface format and forwarded to the user interface.

Example 4 shows such a transaction flow. Protocols are re-entrant and are loaded as required by the various events occuring in the life of the system. A table will give which protocol to use when talking to a given location. The control processes that are created have limited life cycles. There are also numerous processes that could be created using the same protocol when for example many users are performing various provisioning activity on the same network element. When the life cycle of a process is completed, it will merely die out, and all its related data structures will be freed.

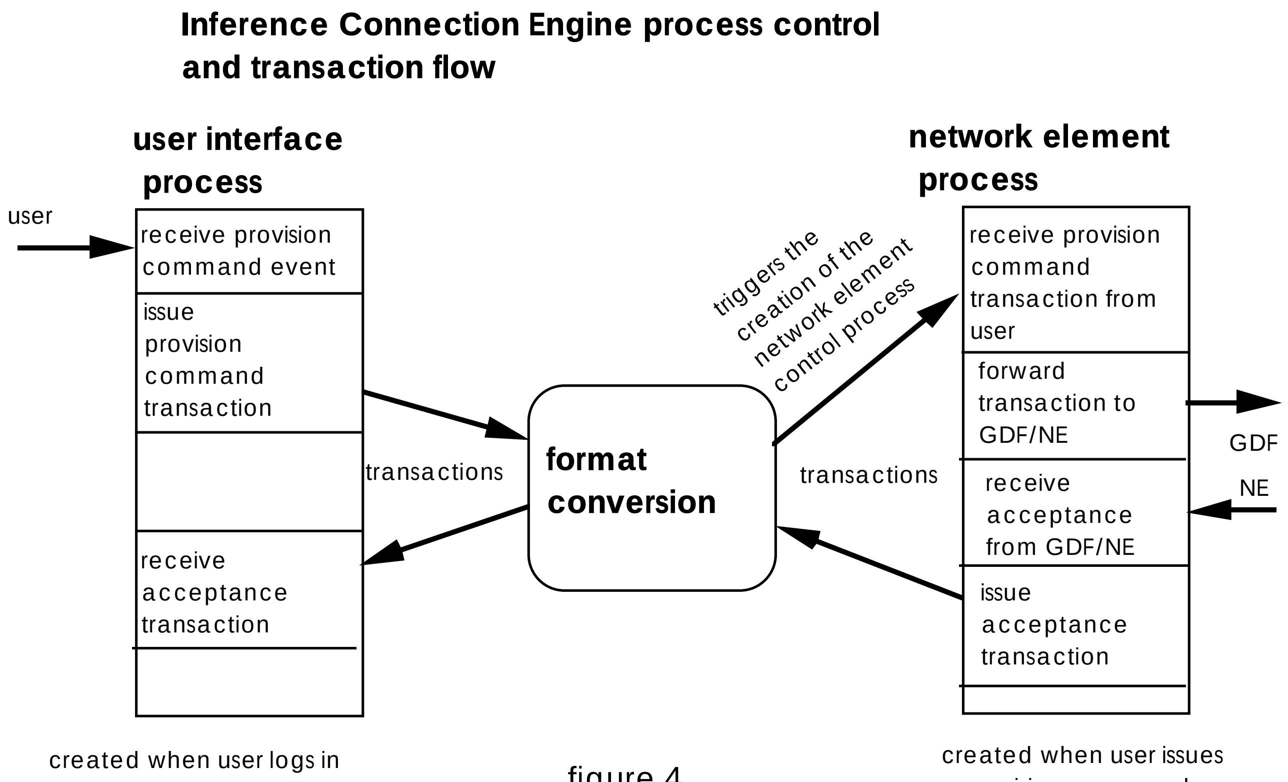

Inference Connection Engine process control and transaction flow
user interface process
user
receive provision command event
issue provision command transaction
receive acceptance transaction
transactions
format conversion
triggers the creation of the network element control process
transactions
network element process
receive provision command transaction from user
forward transaction to GDF/NE
receive acceptance from GDF/NE
issue acceptance transaction
GDF
NE
created when user logs in
created when user issues a provision command

figure 4

Conclusion

The NEC system is in line with the latest concepts of system integration that require systems to be generic and intelligent. The current system is however based on the transaction concept and relies on declarative description languages. While the current system has enabled Bell Canada to explore the realities of Network Element Control and prove its benefits, it became clear that some improvements in the design could be achieved by using more standard description languages such ASN1 for data formats and Formal Description Techniques such as Estelle, LOTOS and other ISO standards for Protocol descriptions.