Elektrotehniški vestnik 72(1): 22–29, 2005
Electrotechnical Review, Ljubljana, Slovenija

Development of MGCP protocol stack for SI2000 digital switch node

Aleksander Vreže, Boštjan Vlaovič, Zmago Brezočnik, Tatjana Kapus

University of Maribor, Faculty of Electrical Engineering and Computer Science,
Smetanova ulica 17, 2000 Maribor, Slovenia
e-mail: {aleksander.vreze, bostjan.vlaovic, brezocnik, kapus}@uni-mb.si

Abstract. In this article, the development of software for SI2000 digital switch node is described, focusing on software architecture for a MGCP protocol stack. A decoder and an encoder for the MGCP protocol have been developed with the PCCTS tool version 1.33, which supports generation of top-down $LL(k)$ parsers in C and C++ programming language. Software planning phase is a very important and time-consuming task, but it is also important to investigate how to implement requested functionality and guarantee proper scalability. To meet the scalability requirements, we had to extend the PCCTS tool with the support for parallel execution of more than one decoder and encoder. Telecommunication management applications also have to support decoding and encoding of MGCP messages. To enable such support for these kinds of applications, we developed an OCX control for the MGCP protocol.

Key words: telephone switch, parser, ANTLR, PCCTS, MGCP, SDL

Razvoj protokolnega sklada MGCP za telefonsko centralo SI2000

Povzetek. Prispevek predstavlja razvoj programske opreme za telefonsko centralo SI2000. Podrobneje je predstavljena arhitektura programske opreme ter razvoj razpoznavalnika in konstruktorja za protokol MGCP, ki se uporablja za krmiljenje medijskih prehodov. Programsko opremo smo razvili s pomočjo prosto dostopnega orodja PCCTS verzije 1.33, ki omogoča generiranje kode v programskih jezikih C in C++. Programsko opremo je treba načrtovati tako, da bo omogočala nadgradnje in da bo skalabilna. Orodje PCCTS smo dopolnili tako, da je omogočeno sočasno izvajanje več razpoznavalnikov in konstruktorjev. Protokol MGCP se uporablja v telekomunikacijskih omrežjih naslednje generacije. Aplikacije, ki se uporabljajo za nadzor perifernih naprav, bodo morale, poleg obstoječih protokolov, podpirati tudi razpoznavanje in tvorjenje sporočil MGCP. Za podporo takšnih aplikacij smo razvili kontrolo OCX za protokol MGCP.

Ključne besede: telefonska centrala, razpoznavalnik, ANTLR, PCCTS, MGCP, SDL

1 Introduction

Prof. A. Graham Bell introduced his "wonderful and miraculous discovery" called "The Telephone" to the general public in the year 1877. Since then the telecommunication market has been changing. In the last decade, changes were greatly influenced by the rapid development of packet-based networks — especially IP networks. Currently, we are facing a great challenge of transition from traditional Circuit Switched Network (CSN) to the packet based Next Generation Network (NGN). Manufacturers and carriers need to adapt to the new technologies to keep up with the ever changing market.

In this paper, introduction of the Media Gateway Control Protocol (MGCP) into the SI2000 digital switch node is presented. MGCP is one of the protocols in the NGN architecture and represents a core of the Media Gateway Controller (MGC) and Media Gateway (MG). A decoder and an encoder for the protocol were implemented in the C programming language*. For the development of the decoder a parser generation tool was used, whereas the encoder was written by hand. To meet scalability requirements, the tool was extended with the support for parallel execution. Additionaly, we developed an OLE Control Extension (OCX) for the decoder and encoder, which can be used in applications for the Windows operating system.

This paper is organized as follows. First, an overview of the MGCP protocol is given. The architecture of the SI2000 digital switch node is presented in Section 3. A basic description of the MGCP software architecture is given. In the next section, a brief overview of the PCCTS tool is shown. In Section 5, architecture of the decoder and encoder for the MGCP protocol is outlined. Finally, a practical example is given. In the conclusion, we comment our work and give directions for further study.

Received 30 January 2004
Accepted 14 February 2005

*The project was partly funded by the Iskratel d.o.o.

2 Media Gateway Control Protocol

The MGCP protocol is used in the communication between MGC and MG (Fig. 1). It is an ASCII-based, application-layer control protocol which can be used to establish, maintain, and terminate calls between two or more endpoints.

The MGCP connection model consists of endpoints and connections. Endpoints represent physical or virtual sources through which data can flow, and connections represent data paths. MGCP endpoints execute instructions received from MGC [1].

MGCP is designed to address the functions of signaling and session management within a packet telephony network. Signalling allows call information to be carried across network boundaries while session management provides the ability to control the attributes of an end-to-end call [2]. Figure 1 presents the flow of a basic call where `User A` calls `User B`.

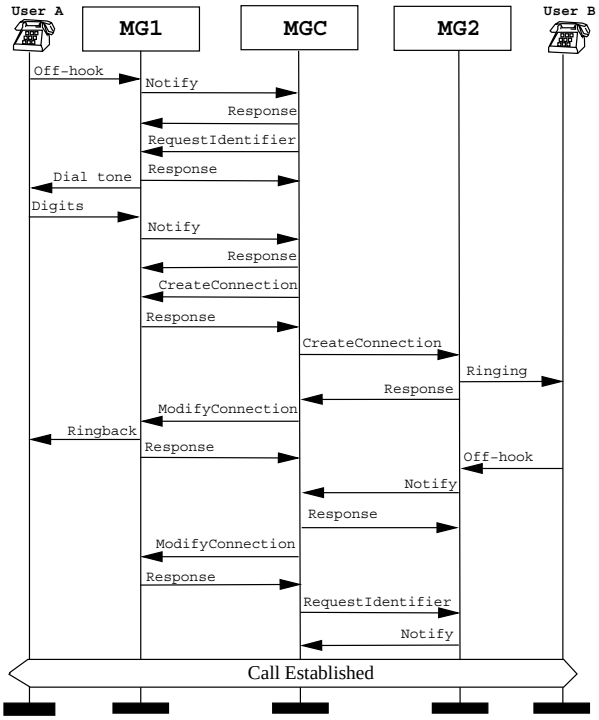

Figure 1. MGCP protocol is used for communication between MGC and MG

2.1 MGCP Commands

The MGCP protocol defines nine commands (Tab. 1), some being sent from MGC to MG and some from MG to MGC (Fig. 1). Each MGCP message is composed of a command line, parameter lines and optionally a Session Description Protocol (SDP) part [3], which describes the connection parameters such as User Datagram Protocol (UDP) port and Internet Protocol (IP) address.

```
Command TransactionID EndPointID Version
(Parameter_name: Value)*

SDP
```

The command line is composed of a command verb, a transaction identifier, an endpoint name and the protocol version. These four items are separated from each other by a space. Parameter lines are composed of a parameter name followed by a colon, a white space and parameter value. In the following lines a valid MGCP message sent from the MGC to the MG1 is shown.

```
CRCX 1209 MG1@uni-mb.si MGCP 1.0
C: 3242342
M: SENDONLY

v=0
c=IN IP4 128.96.41.1
m=audio 3456 RTP/AVP 0
```

Command	Description
EPCF	EndpointConfiguration
CRCX	CreateConnection
MDCX	ModifyConnection
DLCX	DeleteConnection
RQNT	NotificationRequest
NTFY	Notify
AUEP	AuditEndpoint
AUCX	AuditConnection

Table 1. MGCP protocol defines nine commands

3 SI2000 Digital Switch Node

Iskratel SI2000 digital switch node's capacity spans from a few hundred to several thousand ports. It is an advanced modular system which offers basic functionality as well as a wide range of services including PSTN, ISDN, SS7, MGCP, H.323, Centrex, IP Centrex, and SIP-T.

At the moment, a great effort is allocated to the development of the NGN protocol suite with the ambition to support subscribers with packet-based access and smooth transition to NGN-based operation. We will focus only on the development of the MGCP protocol stack. First, a short introduction to the software architecture will be given. Its intention is to prepare the reader for a better understanding of the following sections.

The software provides the functionality, control, and management of the system. Most of it is written in Specification and Description Language (SDL). Various low-level drivers for the peripheral devices and processor-intensive parts of the protocol stack are implemented in C and C++.

SDL has been developed by the switching systems industry and is standardized by ITU-T [4]. It is based on finite state automata, but it uses graphical representation

of flowcharts to show the allowed transitions. Today, several commercial and academic tools with support for SDL are available. We are using Telelogic's ObjectGEODE v4.0 for the academic work and GEODE editor v.2.2.4 for the work on the SI2000 product family. The tool support comprises graphical editing, simulation, code generation, testing, validation, and some verification.

At the highest level of an SDL hierarchical specification there is a *system* object. The system is the entry point to the SDL specification. It comprises a set of *blocks* and *channels*. Blocks can be connected to each other and with the environment by channels. A block is described by sub-blocks or a set of processes. A process is defined by a process graph which usually consists of several pages of state transitions. Processes describe system behaviour. A more detailed description of SDL can be found in [5, 6].

3.1 MGCP Software Architecture in SI2000

Figure 2 shows the architecture of the SI2000 MGCP software modules. It consists of two main parts — SDL and C. SDL describes signalling, signalling control, and connection control. The decoder and encoder of the protocol are implemented in C with the use of the Purdue Compiler Construction Tool Set (PCCTS). Communication between both parts is achieved with the Abstract Data Type (ADT) definitions in the SDL description. They provide access to the Application Programming Interface (API) functions of the MGCP decoder and encoder for the SDL part of the system. We have changed some names of the blocks and processes to improve readability of the paper. The architecture is divided into three levels (Fig. 2):

1. application control level (block `Coco`),
2. protocol control level (block `PrCo`), and
3. TCP/IP control level (block `IPCo`).

The application control level manages Digital Signal Processing (DSP) channels and controls Time Division Multiplexer (TDM) switch connections, whereas the protocol control level executes transactions control and signalling retransmission. The TCP/IP control level manages IP connections (Fig 2).

The `UDP_Mng` process manages the UDP protocol. It receives an UDP packet from the lower layer and forwards an MGCP message to the upper layer, and vice versa. The `PrCo` block consists of two processes. `MGPC_Mng` manages `MGCP_PC` processes and acts as a multiplexer for the communication with the lower layer and external C code (decoder and encoder). `MGPC_Mng` parses the received signal from `UDP_Mng` through the API functions of the MGCP decoder and forwards the received MGCP message to the appropriate `MGCP_PC` process. Since UDP is used as a transport protocol, message flow control is managed by the `MGCP_PC` process. Each MGCP user has

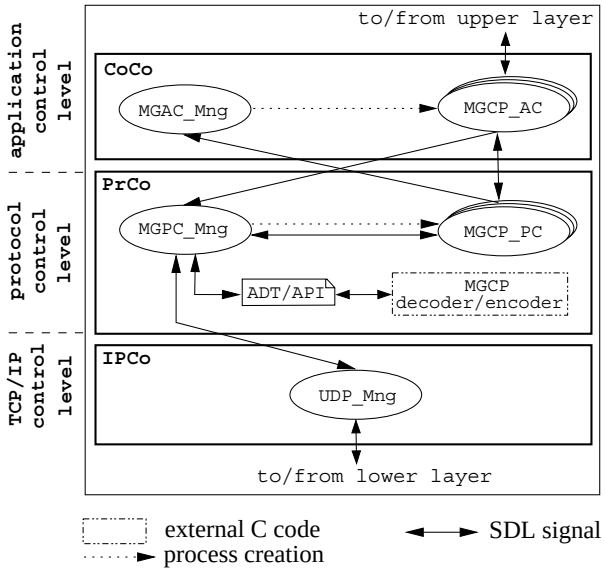

Figure 2. MGCP decoder and encoder integrated in the existing software architecture of SI2000

his/her own `MGCP_PC` process instance (Fig. 2). This decomposition follows recommendations from [1, 7]. After successful reception at `MGCP_PC` the message is forwarded to the `MGCP_AC` process.

The same procedure is followed when the message is created by the system. `MGCP_PC` receives the data from the upper layer (an `MGCP_AC` process). The data are forwarded to `MGPC_Mng`. It creates an MGCP-compliant message with the use of the MGCP encoder. Communication between the `MGPC_Mng` process and the MGCP decoder and encoder is realized by external ADT operators. Next, the message is sent to the `UDP_Mng` process and forwarded to the IP network.

Figure 3 describes process creation and communication between the described processes. Solid lines represent communication between processes by SDL signals while the dotted lines indicate requests for process creation. Figure 3 presents process creation during reception of an MGCP message. The `UDP_Mng` process receives an UDP packet from the IP network and forwards the MGCP message part of the packet to `MGPC_Mng`. If decoding of the message is successful, the `MGPC_Mng` process fills its internal structure and sends it with an SDL signal to an `MGCP_PC` process. If the process does not exist for the current transaction, a request for creation is used. The `MGCP_PC` process checks the received data and forwards them to a peer instance of the `MGCP_AC` process. If the `MGCP_AC` process instance does not exist, `MGCP_PC` sends an SDL signal with the creation request to the `MGAC_Mng` process.

The above example tries to illustrate how a chain of processes is created, when a new MGCP message is received from the IP network. The procedure for the cre-

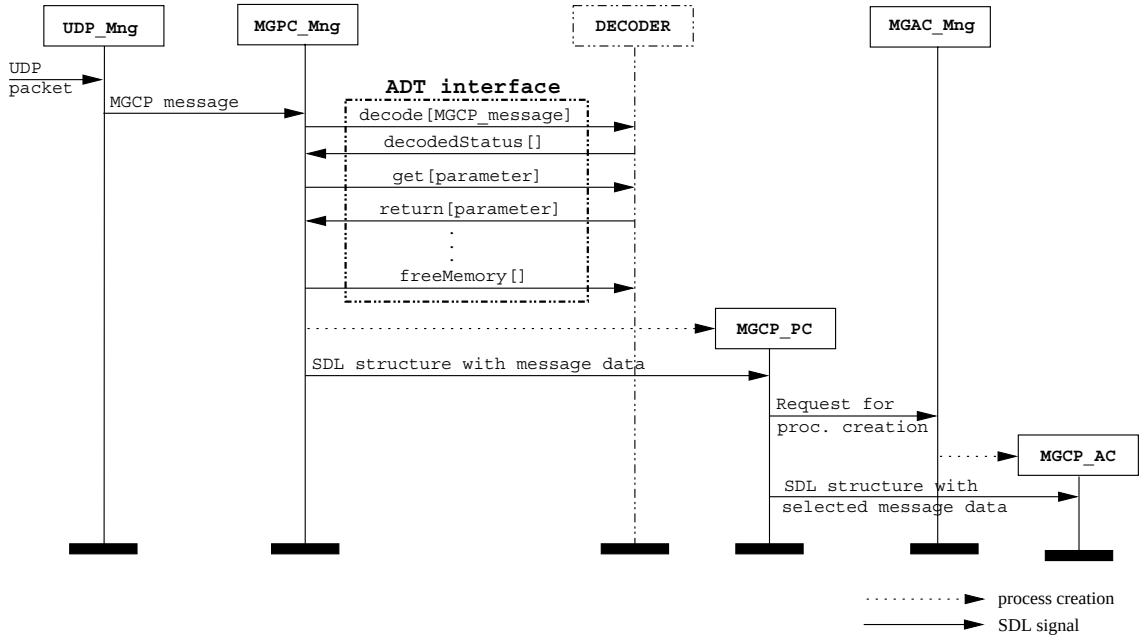

Figure 3. Process communication when an MGCP message is received

ation of the outgoing message process chain is similar.

4 Parser Generator Tools

The decoder for the MGCP protocol was developed with a PCCTS tool. At version 2.0.0, it was renamed to ANTLR. More information about the tool is available in [8].

PCCTS was developed in C. The last released version in 1994 supported code generation in C and C++. The tool supports many kinds of operating systems including Linux and Windows. It is a public domain tool and there are no legal restrictions on its use. Reasons for choosing this tool are: the tool is free, its source code is available, and it includes an efficient error recovery mechanism. One of the project requirements was that the implementation should be carried out in C language. In this section, a brief overview of the PCCTS tool will be given.

4.1 PCCTS Tool

The PCCTS tool version 1.33 includes the following utilities [9, 10]:

1. DLG,
2. ANTLR,
3. SORCERER.

DLG is a tool which produces a deterministic finite automaton for recognizing regular expressions. ANTLR is a tool for creating a parser. It reads a grammar description and builds a set of parsing functions for a top-down parser. The third utility, SORCERER, is a tree parser generator. In our work, DLG and ANTLR were used. PCCTS has the following nice properties:

- it integrates description of lexical and syntactic analysis in one file,
- easy debugging and rich debug information,
- it has a manual and as well as an automatic error recovery mechanism,
- each grammar rule has parameters in return values,
- it accepts grammar in Extended Backus-Naur Form (EBNF),
- the parser generated by ANTLR is human-readable.

PCCTS supports C or C++ interface parsing model.

4.1.1 DFA Lexical Analyzer Generator

The DLG tool reads a lexical description and creates Deterministic Finite Automatan-based a lexical scanner function. The lexical description consists of one or more tokens. Each token starts with the token directive. This may be continued by a token name, which must start with an upper-case letter. This is followed by a regular expression which represents the character string that matches this token (Fig. 4a).

Each regular expression which is part of a token definition may be followed by a *lexical action* enclosed within "<<" and ">>". A lexical action (e. g. `myErrorFunction()`) will be executed every time a lexical analyzer finds the `HEX` token type in the input character string (Fig. 4a). Usually, a lexical action represents a call of one or more C functions. PCCTS defines a number of functions and symbols that can be used. It is also possible for a user to define new functions.

4.1.2 Parser Generator

ANTLR generates a recursive descent parser from a grammar in C and C++ language.

ANTLR grammar description is a collection of rules and actions preceded by a header. A rule is a list of productions, which are separated by symbol "|". Each rule may contain arguments, local variables and return values (Fig. 4b).

```
#token HEX [0-9A-Fa-f]+ << myErrorFunction();>>
```
(a)

```
rule: production_1
            :
            :
    | production_n
    ;
rule[par_1,...] > [ret_1,...];
```
(b)

```
ruleCallID[char *ID]:
                    HEX << strcpy(ID, $1); >>
                    ; << myErrorFunction(); >>
```
(c)

Figure 4. PCCTS overview

The `ruleCallID` contains an argument `ID` and the production `HEX` which represents a hexadecimal number (Fig. 4c). When a lexical analyzer finds the `HEX` token type, the `strcpy` function will be executed. If a lexical error occurs, function `myErrorFunction()` will be executed.

4.1.3 PCCTS Input Format

PCCTS input file ends with "`.g`" extension and may be broken up into many different files. In our work we used C interface parser.

Figure 5 shows a simple example of PCCTS input file. This example illustrates to the reader how PCCTS C programming interface can be used.

```
#header << #include "charbuff.h" >>

<< void myErrorFunction(){
     printf("Error in decoder\n!");
   }
char hexBuffer[100];
main() { printf("Write hex number: ");
         ANTLR(ruleCallID(hexBuffer), stdin);
         printf("Parsed number is: %s\n", hexBuffer);
}
>>

#token HEX "[0-9A-Fa-f]+"
#token "[\ \n\t]+"        << zzskip(); >>

#lexclass START
ruleCallID[char *ID]: HEX  << strcpy(ID, $1); >>
          "@"; << myErrorFunction();>>
```

Figure 5. PCCTS input file — C programming interface

The input file can be divided into four sections. In the first section, header files (`charbuff.h`) can be included. In the second section, user functions (`myErrorFunction`), constants and variables (`hexBuffer`) can be defined. It can also include the main function. In our example we call macro `ANTLR` with two parameters: the first one is a parsing rule and the second one presents the type of input stream (`stdin`). The third section presents the lexical description, which consists of token definitions. A grammar description is defined in the last section. End of the input stream is marked by the "@" symbol. In the next lines a grammar definition for the PCCTS input file is shown.

```
{ #header Action }
( Action | tokenDef | eclassDef )*
( rule | tokenDef | eclassDef )+
( Action | tokenDef | eclassDef )*
```

Generation of the decoder source code is executed in two steps. First, the input file is compiled with the ANTLR tool. Objects of this compilation are some "`.c`" files (depending on the number of input files) and `parse.dlg` file. Next, `parse.dlg` file and some other "`.c`" files are compiled with the DLG tool. Both ANTLR and DLG tools give the user plenty of possibilities. More information is available in [11].

5 MGCP Decoder and Encoder

During the development of the MGCP decoder and encoder RFC 2705 was used as a primary source of the protocol specification. In the current version, we support a required subset of parameters defined in the RFC specification and some proprietary extension parameters. For the final implementation the C language was used. The source code should be short, readable and easy to maintain. The program was designed to run under various operating systems including Linux, Windows, VxWorks, pSOS and HP-UX.

The prepared input file for the PCCTS tool consists of 2294 lines while the generated file contains 6555 lines. The size of the executable file, compiled with `gcc` version `2.95.4` on Linux operating system, is 241 kB. The MGCP protocol standard defines 24 parameters. We implemented 20 of them and some additional extension parameters.

The MGCP decoder and encoder consist of four modules: decoder, encoder, data structures and API functions (Fig. 6).

The decoder was developed with the PCCTS tool. Its function is parsing of MGCP messages. The encoder was developed directly in C language. It creates a valid MGCP message from data stored in internal structures of the encoder. In the data structures, elements of a MGCP message are stored. Each message is its own data structure. The API functions module is divided into two groups

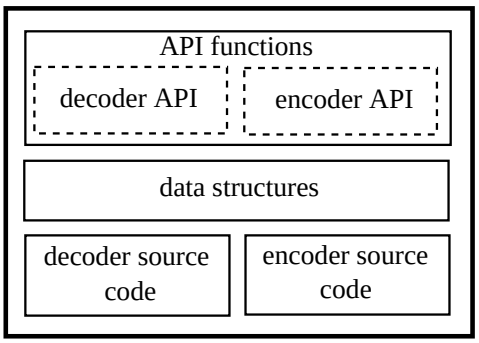

Figure 6. MGCP decoder and encoder modules

(Fig. 6). The first group consists of functions that return value of an individual MGCP parameter, while the second group consists of functions for creating a MGCP message. Each function returns a status of the operation, whereas data are passed as a parameter to the function.

Communication between the SDL processes and the decoder and encoder is achieved by ADT operators (Fig. 2, Fig. 3, Fig. 11). When a user calls the function `MGCP_makeDecode(MGCP_message)`, the decoder parses the MGCP message and fills internal data structures. If an error occurs, the function returns an error status. Each parameter is obtained with a separate call of its own API function. When all parameters are retrieved, the allocated memory has to be realised with a `MGCP_makeClose()` function. Usage of the decoder API functions is shown in Figure 7.

```
/* decode message */
status = MGCP_makeDecode(MGCP_message);
if(status) status = read_CommandType(messageType);
/* read transactionID */
if(status) status = read_TransactionID(transactionID);
/* read other parameters */
          ⋮
/* free allocated memory*/
if(status) status = MGCP_makeClose();
```

Figure 7. Usage of the MGCP decoder API functions

The usage of the encoder is very similar to the usage of the decoder. First, memory allocation and initialisation of the encoder data structures is performed by `MGCP_makeEncode(typeOfMessage)`. Next, internal structures are prepared (Fig. 8). Finally, the construction of a new MGCP message is done with `MGCP_makeString(newMGCPMessage)`. This function returns a correctly formed MGCP message and releases the allocated memory. Figure 8 demonstrates the usage of the encoder API functions.

An MGCP message may include in its body a SDP description. In our implementation, we combined both MGCP and SDP grammar description in the same input

```
/* structure initialisation */
status = MGCP_makeEncode(messageType);
if(status) status = write_transactionID(transactionID);
/* write other parameters */
          ⋮
/* create new message */
if(status) status = MGCP_makeString(newMessage);
```

Figure 8. Usage of the MGCP encoder API functions

file. The PCCTS tool differs from other familiar tools because it integrates the description of lexical and syntactic analysis in one file, which is a very interesting feature, but in the case of longer descriptions of grammar, the input file becomes difficult to read and maintain.

The tool allows to write more than one grammar description in the same input file and supports mechanisms for simple switching between different definitions. At the beginning of the development cycle, we used these features, but it quickly became obvious that this approach has one major disadvantage: the executable file is very long and memory usage dramatically increased.

The tool generates a top down $LL(k)$ parser where k represents the number of lookahead tokens. We set $k = 1$ for many reasons. One of them is that $k > 1$ results in longer generated files and longer executable files.

5.1 Extended PCCTS Tool

For the implementation of the MGCP decoder and encoder we used the PCCTS tool written in C. As it uses global variables, it does not support parallel execution. In our work, we implemented the support for parallel execution by extension of the tool. The tool extension was a hard and time-consuming task. The tool extension consisted of: source code analysis, variable determination and error recovery.

First, we analysed the source code and determined, which variables were changed during the parsing of the message with a Data Display Debugger (DDD). Next, we exported all these variables in a new file with the `#export` directive in C. The exported variables were changed by adding one dimension. For example, variable `int var` became array `int var[]`. Additionally, we had to extend some functions and macros.

Our extension supports execution of a parser generated with $k = 1$ because we did not need to support $k > 1$. Our solution is not universal for $k = n$, therefore in the case of need to achieve $k = 2$, we would have to extend some additional variables and functions. Figure 9 shows the architecture of the program during the parallel execution. Each thread of the decoder and encoder has its own data structure with decoded and encoded data.

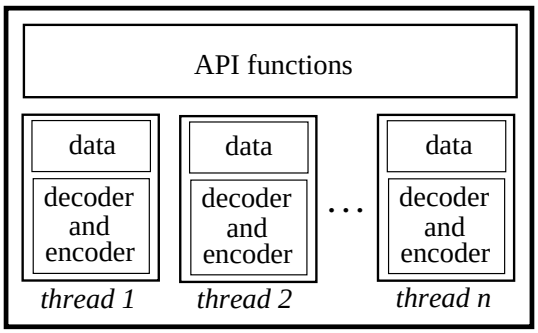

Figure 9. Decoder and encoder data structures during the parallel execution

5.2 MGCP OCX Control

The OLE Control Extension (OCX) control is an independent program module which can be used by the developer of applications in a Windows environment. OCX controls end with an "`.ocx`" extension. They represent Microsoft's second generation of control architecture, the first being VBX controls written in Visual Basic.

The source code for the encoder and decoder for MGCP protocol was written in C. This makes it easy to port and compile in various operating systems including Linux and Windows.

To support quick inclusion of the MGCP decoder and encoder, we developed OCX control. The MGCP OCX control can be easily used by developers of Windows applications. It can be used for prototyping, testing and management applications which must support decoding and encoding of MGCP messages.

The OCX control was implemented in Microsoft Visual C++ 6.0 SP5. We used the existing MGCP decoder and encoder source code and additionally implemented an OCX API interface. The size of the compiled MGCP OCX control is 548 kB. Figure 10 shows the architecture of the OCX control for the MGCP protocol.

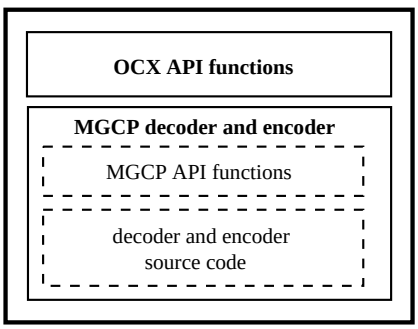

Figure 10. MGCP OCX program structure

We developed an environment for testing the implementation of the MGCP protocol stack where we included the MGCP OCX control. The program was developed with Visual Basic 6.0 SP5. It supports a lot of functionality including a graphical editor for creating a MGCP message.

6 Practical example

In this section we present how the MGCP stack encodes a new `Notify` (Tab. 1) message. In Figure 11 communication between the processes and the encoder is shown. Process creation and communication is similar to the opposite direction as shown in Figure 3. Process `MGCP_AC` receives a request for encoding a new message. First, it sends a request for a peer process creation to the `MGPC_Mng` process, which creates a new instance of the `MGCP_PC` process. Next, `MGCP_PC` receives an SDL signal with message data from `MGCP_AC` and forwards it to `MGPC_Mng`. It allocates and initializates the encoder internal data structures with the ADT operators. Then, it fills the structures with message elements. Finally, a new MGCP message is encoded by calling the `MGCP_makeString` API function. `MGPC_Mng` forwards the new MGCP message to the `UDP_Mng` process (Fig. 11). `UDP_Mng` receives the SDL signal with the message string (`Notify` message). It creates a new UDP packet, which includes the MGCP message string and sends it to the IP network.

7 Conclusion

In this article, a development of the MGCP decoder and encoder for the SI2000 digital switch node is presented. We developed the decoder and encoder in a Linux operating system. The program has been ported and compiled in various operating systems including VxWorks and pSOS. To support parallel execution of the decoder and encoder, we extended the PCCTS tool. To support quick integration of the MGCP encoder and decoder, OCX control was developed.

The decoder and encoder succefully run in the SI2000 digital switch node. The program is very stable. In the future, it will be integrated in various telecommunication devices including terminal adapters and telephones.

We will also extend the program with other specified parameters and some of our own parameters. Architecture of the software is modular and will therefore be easy to maintain and upgrade.

8 References

[1] M. Arango, A. Dugan, I. Elliott, C. Huitema, S. Pickett, "MGCP: Media Gateway Control Protocol, RFC 2705, Network Working Group," October 1999.

[2] D. Collins, *Carrier Grade Voice Over IP*. New York: McGraw-Hill Companies, Inc., 2001.

[3] M. Handley, V. Jacobson, "SDP: Session Description Protocol, RFC 2327, Internet Draft," 1998.

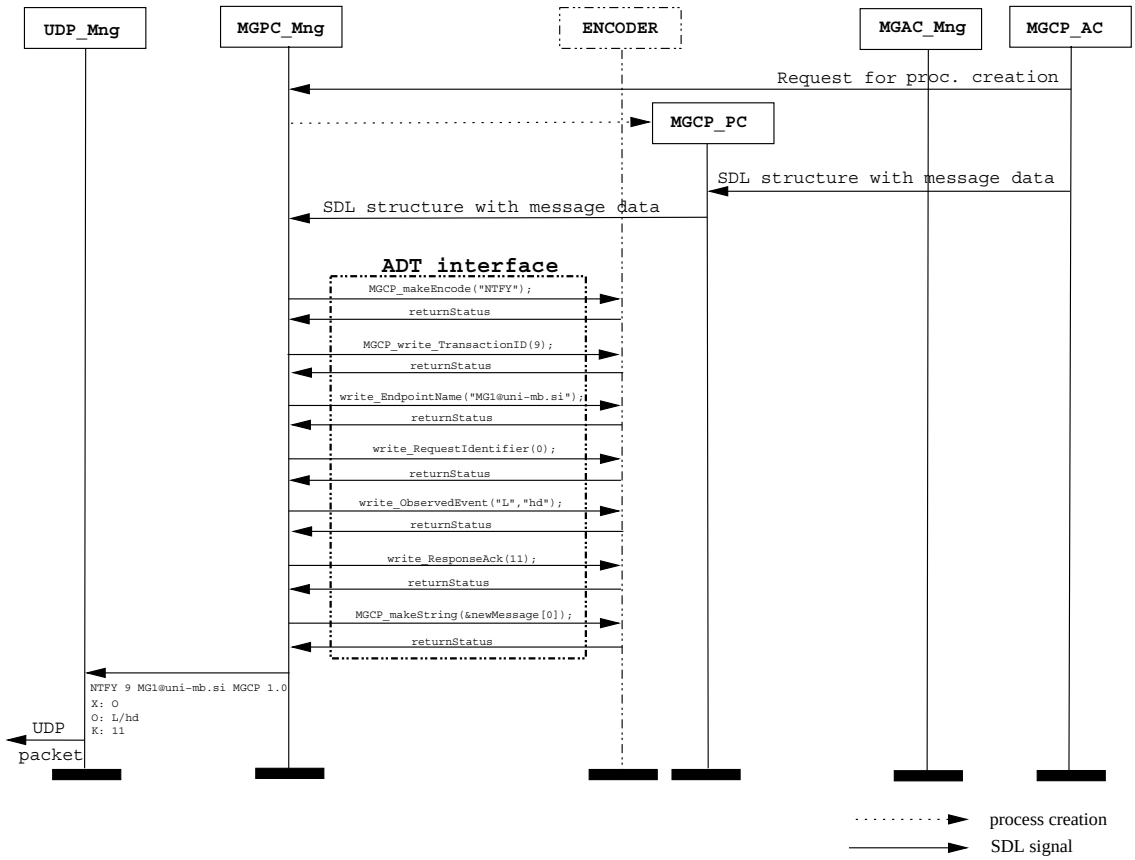

Figure 11. Example of encoding the `Notify` MGCP message

[4] ITU-T Recommendation Z.100, "SDL with applications from protocol specification," tech. rep., CCITT specification and description language (SDL), 1993.

[5] J. Ellsberger, D. Hogrefe, A. Sarma, *Formal Object-oriented Language for Communicating Systems*. Prentice Hall Europe, 1997.

[6] B. Vlaovič, Z. Brezočnik, "Analog subscriber call generator," *Electrotehnical Review*, vol. 69, no. 5, pp. 259–265, 2002.

[7] International Softswitch Consortium, *MGCP Implementation Guide*, April 2000.

[8] ANTLR home page, `http://www.antlr.org/`.

[9] T. J. Parr, *Language Translation Using PCCTS and C++*. Automata Publishing Company, 1993.

[10] T. J. Parr, R. W. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," in *Software - practice and experience*, vol. 25 of *7*, 1995.

[11] T. J. Parr, H. G. Dietz, W. E. Cohen, *PCCTS Reference Manual*, August 1991.

Aleksander Vreže (Student Member, IEEE) received diploma in Computer Science from Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 2001. He is in his third year of a Ph.D. studies at the same faculty and works as a researcher in the field of telecommunications. His main research interests cover voice communications over packet-based networks and parallel algorithms. His current research includes parallel algorithms for binary decision diagrams.

Boštjan Vlaovič (Student Member, IEEE) received diploma in Electrical Engineering from Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 1999. He is a Ph.D. candidate at the same faculty and works as a researcher in the field of telecommunications. His special interests cover voice communications over packet-based networks and their integration with traditional PSTN. His current research interests are focused on formal verification, especially model checking.

Zmago Brezočnik (Member, IEEE) received M.Sc. and Ph.D. degrees from the University of Maribor, Faculty of Electrical Engineering and Computer Science, in 1986 and 1992, respectively. He is a full professor, head of Laboratory for Microcomputer Systems, and vice dean of education at the same faculty. His main research areas are formal hardware and protocol verification, especially symbolic model checking and binary decision diagrams.

Tatjana Kapus (Member, IEEE) received M.Sc. and Ph.D. degrees from the University of Maribor, Faculty of Electrical Engineering and Computer Science, in 1991 and 1994, respectively. She is currently an associate professor there. Her primary research interests are in the area of formalisms and tools for specification and verification of reactive systems, such as, for example, communication protocols.