Third Int. Workshop on GNs, Sofia, 1 Oct. 2002, 59-64

IMPLEMENTATION OF GENERALIZED NETS WITH EXTERNAL MODULES

Trifon Trifonov[1] and Valentina Radeva[1]
[1] CLBME - Bulg. Academy of Sciences, Acad. G. Bonchev Str., 105 Block, Sofia-1113, Bulgaria,
e-mails: trifon.trifonov@mail.clbme.bas.bg,valia@vip.bg

Abstract: The aspects of connecting external modules to Generalized Net models are discussed. Some interface functions for running external programs are presented on the basis of the GN simulator. An example of a GN model using external modules is defined and implemented.

Keywords: Generalized net, external module, document reading, optical character recognition

1. INTRODUCTION

Generalized Nets (GNs) (see [1]) have already proved to be a very useful modelling instrument. They have been used to model other formalisms - expert systems ([2]), machine learning ([3]), neural networks, UML diagrams ([4]), etc. The main reason for the wide application of GNs is the flexibility of their definition. Despite some restrictions, which prevent conflicts and deadlocks, components such as predicates and characteristic functions allow dependence not only on events in the GN itself, but on outside events. The environment can be monitored with external sensors, which can do computation and then return the result to the GN. Of course, the calling of an external module can be a highly time consuming task. This should be kept in mind, when constructing such a model. An example is mentioned in [1], where the logic value of a predicate in a transition depends on the truth of the Fermat's last theorem. As we know, mathematicians needed over 300 years to solve this problem. Therefore, this model cannot be used, when we concern times such as days or months. In order to make such a model usable, the GN time scale should be changed so that a tick of the GN model life equals to a large real time period.

2. ADDRESSING EXTERNAL MODULES FROM A GN

We will consider two kinds of addressing external modules - synchronous and asynchronous. We have synchronous addressing, when the GN module calls an external module and waits for it to finish. As far as we like to receive a result from module, an external sensor, for example, we should use synchronous addressing. The synchronous addressing is time critical, because the execution of one module delays the whole system. One variant of synchronous addressing is the periodic addressing. Such occurs when a sensor needs to be addressed at a specified frequency in order to work correctly. As an example we may consider a GN, which keeps statistics of the weather forecast on a certain radio station. The weather bulletins

are broadcasted in one hour intervals and continue one minute. The GN should tune to the specified frequency every hour at the precise time, otherwise it would miss the forecast. Also, it should be able to wait for a minute to listen to the whole forecast.

We have non-periodic addressing, when a module can be addressed at any time. An example is an external sensor as the thermometer. Undoubtedly, it is harder to model periodic addressing of external modules, as it is time critical. That means, it should be almost impossible to mix it with a non-periodic addressing, which can delay the execution of a periodic one. Although, periodic synchronous external modules can be included in GNs, because the GN works on time ticks. The only thing to synchronize are the GN time scale and the module time scale.

The limitations of the periodic synchronous modules are not suitable for some complicated tasks, where the external module is unreliable on execution time. An example is an expert decision, where we have to wait enough to receive an answer. Therefore, in this article we will concentrate on implementation of non-periodic synchronous external modules. Thus we omit many technical hardships in the implementation.

The other type of addressing - asynchronous addressing, is used whenever we call an external module and want to receive a partial result almost immediately. Then the external module can continue working and be ready with the full result later. If we need the full result, we can check it later by calling the module again. Note, that this kind of addressing does not delay the GN model and is useful for parallel task execution, especially when the external program is highly time consuming. As an example, we may have a statistic module, which we provide with numerical information and from which we expect approximation results. It can be addressed asynchronously and the more time we wait for the module to work, the better the approximation will be.

3. CONNECTING EXTERNAL MODULES TO A GN MODEL

Before we proceed to the particular implementation of external modules, we will consider the theoretical possibilities of connecting external modules to GN. If we look closely at the definition of the Generalized Nets in [1], we will see that we have two different possibilities. First, we may include a predicate in an index matrix, which depends on some external condition, which can be determined by an external sensor. Here we have two different types of sensors. The first type is the thermometer type - a sensor performs a measurement or a computation and returns a value, that can be later used in a condition. For example:

$$W_1 = \text{“}temperature < 25C\text{”}.$$

The second type is the decision type. Here we give an example of an external expert system, which has to answer “Yes” or “No” to a certain question. For example:

$$W_2 = \text{“it is going to rain today (according to the weather forecast)”}.$$

Since the predicates in a whole row of the index matrix are evaluated any time a token is waiting in an input place, we are not able to put synchronous addressing here, because the order of predicate evaluation is not specified theoretically and is considered an internal task of the particular GN implementation. On the other hand, when we include asynchronous addressing, we should note the external tasks should be completed in time, less than a GN time tick. In other words, we should prevent abnormalities of the “Fermat’s theorem” type, already discussed in Section 1.

Another possibility of connecting external modules is using the characteristic functions. These functions are assigned to a place and they define new characteristics of entering tokens. Some of these functions can be external functions. Since a token enters only one place at a moment of the GN time, external function calls can be synchronous, which is an advantage over the previous possibility. Here we also have two possible variants. First, we have a function, which calculates something using its parameters and returns a result. An example can be some differential equation, which is solved with an external program for solving such equations. The equation and all additional conditions are passed as parameters and the solution is returned as a result. The other type of external function is the procedural type of function. This is a function in which we care about the side effects of the execution. Such procedure does not return a result, or returns only "Success" or "Failure". Here we have external effectors rather than external sensors. They perform specific tasks, which the GN as a formalism cannot perform. This is the kind of connecting to external modules which is most useful and easy if technical implementation regarded. We will discuss it later in a more detailed example.

4. A PARTICULAR IMPLEMENTATION OF GNs WITH EXTERNAL MODULES

In the implementation discussed here, we used as a basis the GN simulator "Sim2000" [5]. It was developed using Borland Delphi 5 and is targeted for the Win32 architecture. We decided to use only the connection through characteristic function, as the more natural between the two possibilities. When evaluating characteristic functions, the simulator checks for an internal function to execute, and if none is found, then it sends a message for an external function call. We have implemented a gateway for executing external Win32 processes, by adding the external *exec()* function. The syntax is:

exec(⟨executable-filename⟩,⟨command-line-parameters⟩), where

⟨*executable-filename*⟩ is a string, containing the filename of the targeted executable with the full path,

⟨*command-line-parameters*⟩ is a string, consisting of the parameters, passed to the process as command line parameters.

This function executes a Win32 process. Then it loops, waiting it to finish. The *exec()* function returns as result the integer "1" on success and "0" on failure. Although the process does not need to be started at a specific moment, it should be waited for to finish. That is why the *exec()* function performs synchronous execution.

According to the specifics of the Win32 shell, an asynchronous extension of the function has been implemented. The syntax of the *shell()* function is:

shell(⟨filename⟩,⟨command-line-parameters⟩), where

⟨*filename*⟩ is a string, containing the filename of a document to be opened or an executable to be run with the full path,

⟨*command-line-parameters*⟩ is a string, consisting of the command-line parameters.

The *shell()* function passes a document or executable for opening by the Win32 shell. The result, like in the *exec()* function, is "1" on success and "0" on failure. The main difference between *exec()* and *shell()* is that the last function does not wait for the process to finish, i.e. it is executed in the background. Also, *shell()* is also capable of executing documents - e.g. texts, image, video, audio.

The functions we have described both act as the "procedural" type of functions. They

accept parameters, perform an action, but return no result. The only way to get a result of the execution of such external modules is to use an external mediator, such as a disk file. Such approach is applied in the example, discussed in the next section.

In order to implement the "functional" type of calling external modules, it is suitable to define a protocol, by which external functions return results. At this time we have restricted ourselves only to executing Win32 console applications and to consider their output as a string result. For such purposes a *console()* function has been developed:

console(⟨executable-filename⟩,⟨command-line-parameters rangle), where

⟨*executable-filename*⟩ is a string, containing the filename of the targeted Win32 console executable with the full path,

⟨*command-line-parameters*⟩ is a string, consisting of the parameters, passed to the executable as command line parameters.

The output is gathered in a temporary file and returned as a result of type string.

The three described functions give enough flexibility to define GN models in "Sim2000", which use external modules. In the following section we describe an example, that displays the application of such GN models.

5. A GN MODEL OF AN INTELLIGENT SYSTEM FOR MACHINE READING OF TRADE DOCUMENTS

In [6] a GN modelling a system for machine reading of trade documents and extracting structured data was presented and discussed in details.

We will define a new GN model, based on the one from [6] and we will show the abilities of "Sim2000" to simulate the system behaviour by calling external programs on each step of the process.

$$E = \langle\langle\{Z_1, Z_2, Z_3, Z_4\}, \pi_A, \pi_L, c, f, \theta_1, \theta_2\rangle, \langle\{k\}, \pi_K, \theta_K, \rangle, \langle T, t^0, t^*\rangle\langle X, \Phi, b\rangle\rangle$$

$$Z_0 = \langle\{l_0\}, \{l_1\}, t_{0,1}, t_{0,2}, r_0, m_0, \square\rangle$$

$$Z_1 = \langle\{l_1\}, \{l_2\}, t_{1,1}, t_{1,2}, r_1, m_1, \square\rangle$$

$$Z_2 = \langle\{l_2, l_3\}, \{l_3, l_4\}, t_{2,1}, t_{2,2}, r_2, m_2, \square\rangle$$

$$Z_3 = \langle\{l_4, l_5, l_6\}, \{l_5, l_6, l_7\}, t_{3,1}, t_{3,2}, r_3, m_3, \square\rangle$$

$$Z_4 = \langle\{l_7, l_8, l_9, l_{10}\}, \{l_8, l_9, l_{10}, l_{11}\}, t_{4,1}, t_{4,2}, r_4, m_4, \square\rangle$$

$$r_0 = \begin{array}{c|c} & l_0 \\ \hline l_1 & true \end{array}$$

$$r_1 = \begin{array}{c|c} & l_1 \\ \hline l_2 & true \end{array}$$

$$r_2 = \begin{array}{c|cc} & l_2 & l_3 \\ \hline l_3 & true & false \\ l_4 & false & true \end{array}$$

$$r_3 = \begin{array}{c|ccc} & l_4 & l_5 & l_6 \\ \hline l_5 & true & false & false \\ l_6 & false & true & false \\ l_7 & false & false & true \end{array}$$

$$r_4 = \begin{array}{c|cccc} & l_7 & l_8 & l_9 & l_{10} \\ \hline l_8 & true & false & false & false \\ l_9 & false & true & false & false \\ l_{10} & false & false & true & false \\ l_{11} & false & false & false & true \end{array}$$

We will not fully describe all other GN components, since they are not important for our implementation.

The document is represented as a token, going through the GN and changing its characteristics. The main characteristics used are:

Program - the filename of the external program to be executed on the next step

InputParam - the filename of the input file for the program

OutputParam - the filename of the output file for the program

OtherParams - additional program input parameters

There are other initial characteristics, such as program paths, additional options and so on, which can be changed to give the process flexibility. Instead of strict definition of the characteristic functions, we will give informal description of each step of the process.

l_0 - "The document enters the net"

l_1 - "Recognize characters in the document (OCR)"

l_2 - "Joining characters to words"

l_3 - "Spellcheck the words in the document"

l_4 - "Document type recognition and extraction of structured data based on the data layout knowledge associated with the document type"

l_5 - "Perform second character recognition with a restricted character set"

l_6 - "Coerce characters into the appropriate type with equivalent replacements"

l_7 - "Correct user-specified fields of specific interest, which are represented in a database"

l_8 - "Prepare user-specified fields for futher interptetion"

l_9 - "Interpret user-specified fields of specific interest, which are represented in a database"

l_{10} - "Verify user-specified fields of specific interest, which are represented in a database"

l_{11} - "The document exits the net"

After the token leaves the net, its **OutputParam** characteristic contains the filename, where the final result is stored.

The definition of the net is entered into the "Sim2000" simulator as a file in the Generalized Net Language. We will not give the entire definition, but just a description of one of the characteristic functions:

SpellChecker:

Result = exec(Program, concat(" -i=",concat(InputParam,concat(" -o=", concat(OutputParam,OtherParams))))),

Program = "WfSpellchecker.exe",

InputParam = OutputParam,

OutputParam = concat(P1 , concat ("\spell\", concat(P2,"_sp_words.xml "))),

OtherParams = concat(" -dict="", concat(SYS_F2X, concat("dictionaries\", concat(KEYWORD_DICT,"" " -limit=0.5 -full_check=WFTRUE")))).

Here we make use of the internal function *concat()*, that concatenates two strings. The **Result** characteristic holds the result from the previous program execution. Here, the *exec()* function is used, thus using a "procedural" call. The information about the document is stored on several external disk files. The token has only their filenames as characteristics.

This model is working and has been tested on real documents.

6. CONCLUSION AND FUTURE WORK

We have discussed how the connections of external modules to GN models can be performed. We showed that two approaches to that problem exist – using predicates and using characteristic functions. We chose the second approach, since it has the possibility of synchronous addressing of external modules and seems more natural. We have developed three different functions to execute external Win32 programs - *exec()*, *shell()* and *console()*, which implement respectively the "procedural" type of execution with synchronous and asynchronous addressing and the synchronous "functional" type of execution. Note that the method, used in the *shell()* function cannot be applied to receive asynchronous "functional" type of execution, since the function has to return a value immediately, so we have to wait for the function to finish. That makes the execution synchronous.

A topic of future work will be advancing the "functional" approach by defining a more strict protocol for returning results from external functions and extending the range of external modules, which can be used by a GN model. These can include functions in Win32 dynamic-linked libraries (DLLs), connections to mathematical software as Mathematica or Matlab, using Component Object Model (COM) objects, Prolog-based expert systems, functional programming languages. The goal is to make the GN model application wider, using contemporary development tools. Another purpose is to develop more GN models, using external modules in order to see using which particular external tools are most useful in GN development.

REFERENCES

[1] Atanassov K., Generalized Nets, Singapore, World Scientific Publishing, 1991.

[2] Atanassov K., Generalized Nets in Artificial Intelligence. Vol. 1: Generalized Nets and Expert Systems, Sofia, "Prof. Marin Drinov" Publishing House of Bulgarian Academy of Sciences, 1998.

[3] Atanassov K., H. Aladjov, Generalized Nets in Artificial Intelligence. Vol.2: Generalized Nets and Machine Learning, Sofia, "Prof Marin Drinov" Publishing House of Bulgarian Academy of Sciences, 2000.

[4] Koycheva, E., T. Trifonov, H. Aladjov, Modelling of UML sequence diagrams with generalized nets, Varna, First International IEEE Symposium "Intelligent Systems", 10-12 September 2002, 79-84.

[5] Aladjov H., Nikolov N., Georgiev P., Atanassov K., Software for generalized nets. Annual of Technical University, Sofia, Vol. 50, No. 3, 1999, 125-132 (in Bulgarian).

[6] Radeva, V. An Intelligent System For Machine Reading Of Trade Documents. in the present Proceedings, 7-13.