Formal Model, Language and Tools for Design Agent's Scenarios in Call Center Systems

Nikolay Anisimov[*], Konstantin Kishinski, Alec Miloslavski,

Genesys Telecommunication Laboratories, Inc.
1155 Market Street, San Francisco, CA, USA 94103
Fax: (415) 437 1027
E-Mail: {anisimov,kotc,alec}@genesyslab.com

Abstract

During the last few years there continues to be remarkable growth in telephone call-center systems. There are many applications of call-centers in different areas of business such as in telemarketing, insurance, customer service, electronic commerce, etc. Moreover, in some cases it is reasonable to think of a call-center as an integrated part of a whole business system responsible for the telephone interface with the outside world. Typically, a call center consists of a set of operators, called agents, who process inbound calls from clients. Call processing may involve the use of computer systems (e.g. database), other devises (e.g. fax-machines, interactive voice response units) as well as communication with other agents (e.g. deliver a call to more qualified agent, making a consulted call). The call processing may also produce outbound calls. The treatment of each call being processed is heavily regulated by scenarios called scripts which are specially designed for specific kinds of the calls. The design of such scripts is one of the main problems in call center maintenance. To cope with this problem we need special tools, i.e., scripting language, corresponding editor, related environment. In this paper we present an ongoing project aimed at the design of such a platform. We introduce a Petri net-based model for formal representation of scripts and a logical structure of the call center. The model, called script-net, is based on object-oriented Petri net dialect belonging to a class of high-level Petri nets. In particular, the model allows one to formally represent scripts, their communication with agents and other resources, exception handling, time constrains. We also consider some implementation issues. In particular, we outline a visual iconic language specially designed for script specification. The semantics of the language is based on script-nets. An agent of the call center can be perceived as a specific resource and is implemented with the aid of Internet/Intranet technology. To illustrate the use of suggested tools, some typical examples of scripts are presented including scenarios for inbound and outbound telemarketing.

1. Introduction

Over the last few years phone call-center systems have continued to grow at a remarkable pace. Several manufacturers and service providers are developing and introducing systems with enhanced functionality, principally through what is known as computer-telephony integration (CTI) [15]. The general purpose of a call center is to connect operators called *agents* with members of the public called *clients*, i.e., people interested in using the services of the call center. Typically a call center is based on at least one telephony switch to which agent stations are connected by extension lines and directory numbers, and to which incoming and outgoing trunk lines may carry telephone calls between the switch and the parties who call in. In addition, most modern high-capacity call centers have agent stations that include computer platforms, often PCs, equipped with video display units (VDUs). The PC/VDU platforms are typically interconnected, usually by a local area network (LAN). There may also be servers of various sorts (e.g. data base or fax server) for various purposes on the LAN, and the LAN may also be connected to a CTI server, in turn connected to the central switch through a CTI link.

[*] Corresponding author.

Within a call center, agents process telephone calls from clients and carry out call-related business.

The typical processing of calls includes using data from computer systems, including databases; incorporating other devices such as fax and e-mail; and communication with other agents. The communication of the agent during call processing is heavily directed by a specific scenario, specially developed for such calls by telemarketing experts. These scenarios are referred to as *scripts*. The same agent can work with varying call types, controlled by different scripts. Thus, a call center is a distributed system, usually built on top of a local area network that connects agent stations, server computers and telephone equipment.

It is interesting that concept of workflow management [1,6] can be very useful in designing call centers. In fact, a call center can be understood as a specific case of a workflow system, which substitutes telephone calls for documents circulating in the system. We should also mention that because office activity very often involves working with inbound and outbound calls, the processing of such calls should be naturally incorporated into workflow management systems.

The present paper is devoted to call center management, and is specifically directed toward scripting for call centers. Usually, scripts are written in a relatively high-level programming language. The complexity of a call center presents a challenge for any programming tool. Moreover, as with any other sort of programming, when a bug appears or a change is made in the purpose or operation either of a call center or a segment of call center operations, it is often necessary to rewrite a large number of scripts. This endeavor is no small task, and may take a considerable time. Moreover, such reprogramming introduces numerous opportunities for errors, both in programming and in the layout of the script.

Given the nature of call center management, and scripting in particular, it is highly desirable to reduce the complexity and amount of effort required to direct these activities. It is especially important to simplify the activities of agents, such as engagement with clients, and to provide enough flexibility so that changes and adaptations can be easily and quickly made without fear of error. To handle this issue we need special tools, such as a visual language, graphical editor, and others. Essentially, the requirement is to build a platform for a generation of CTI applications of varied types. Such tools for the generation of CTI applications already exist, but for the most part they do not take into account the distributed nature of call centers and therefore do not allow the production of scripts with complex communications between agents, hardware and other resources. In this paper we present the progress we have made in an ongoing project aimed at designing such a platform.

If we examine scripts of a typical call center, we see that their key features include the flexible use of resources during call processing; extensive manipulation of calls, including attached data; allowance for exceptions; complexity of real-world scripts; parallel call processing; and strict requirements for real-time call processing. It is clear that scripting tools should be designed according to a formal approach. This paper is devoted to developing such an approach, using the theory of Petri nets [11,12]. More specifically, in this paper we build a Petri net-based formal model for representing call center scripts.

2. Structure of a call center

In this section we present an abstract model of a typical call center that will serve as a subject for the formalization process. From now on, the call center will be referred to as a "system ".

Typically, a system operates with a set of resources. These are: equipment (e.g. phones, fax machines, switches, a local area network, etc.), software components (database, text editor, etc.) and personnel involved in system operation (agents, administration). All communications of the system is accessed through these resources. A typical Call-Center environment is shown on Figure 1.

From the point of view of applications, the system can be perceived as a collection of communicating objects. We will divide all objects of the application level into two types: resource objects and call objects. The former represents objects corresponding physical resource of the system while the latter represents objects intended for call processing.

The behavior of each object is regulated by a scenario specification, called a `script'. There may be several objects working in accordance with one and the same script. For example, for a script describing the behavior of a telephone, there may be several objects corresponding to actual telephones in the system; a script specifying the call processing, may have several objects processing different calls of the same type.

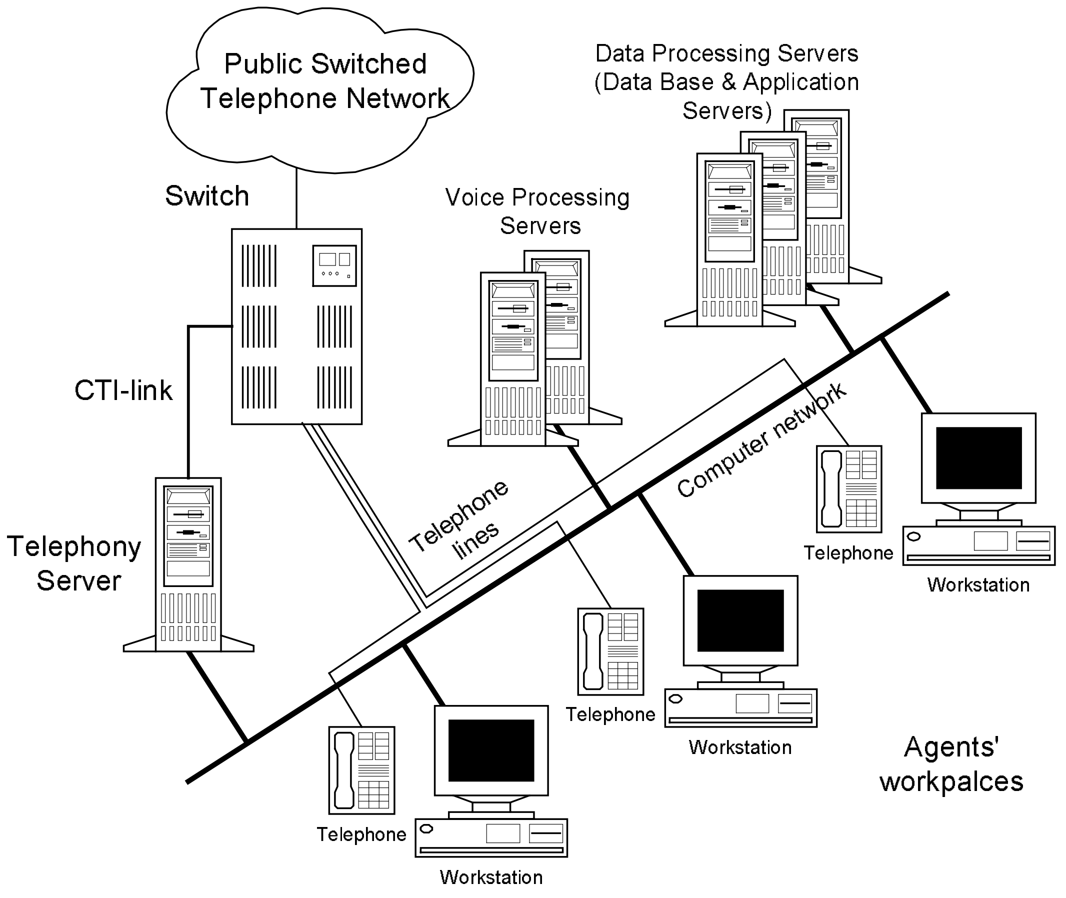

Figure 1: A call center environment

We will assume that each script is identified by a unique name within the system. Moreover, we associate with each script a domain of object names, to identify each object within the script. This addressing scheme allows us to uniquely identify objects within the whole system.

3. Formal Model

For formal specification of scripts of CTI-applications, we developed a Petri net-based model called *script-net* [3] combining some featured from other Petri net models [11,12]. The model consists of the following four (quite orthogonal) constituents:

1. High level and object oriented Petri net model called *cooperative nets* [14] that allows to represent complex system as a set of subsystems communicating via the client server protocol [13];

2. For structured specification of complex scripts, we suggest the concept of hierarchical transition [8]. Under this concept, a net can be represented as a set of disjointed subnets with links between hierarchical transitions and subnets forming a hierarchical structure. Firing any hierarchical transition results in execution of its internal net. This construction makes script representation modular and allows for the simple modification and reuse of specifications.

3. For processing exceptions in scripts, such as receipt of unsolicited events, we suggest a macroplace construction [2].

4. To represent real-time constraints that are very critical for scripts, we incorporate Merlin's time constructs into our model [10].

In the following we consider some of these constituents in detail and show how they can be used.

3.1. Basis of script-nets

As a basis of script-net we take an object-oriented model of high level Petri nets known as *cooperative nets* [14] that allows us to represent a system as a set of communicating subnets. In particular, each transition can be associated with the process of communication (sending or receiving of message) with other script-nets:

- Sending a command: ***s(script(v).com(v_1, ...,v_n))***, where ***script(v)*** specifies a target object, and ***com(v_1,...,v_n)*** is a command with parameters.

- Receiving a command: ***r(script(v).com(v_1, ...,v_m))***, that gives a command with parameters from the object ***script(v)***.

This enables us to specify a communication between a CTI application and a server using a client-server protocol [13]. Moreover, it is possible to associate a transition with a creation of new objects for some script-net.

- Creating a new object: ***c(script(v).v_1,...,v_k)***, where ***script(v)*** identifies a creating object and ***v_1,...,v_n*** its initial parameters.

Thus each script of a CTI application can be described as a corresponding script-net. In this case the process of call processing can be understood as creating an object (injecting a token in head place of script net) and moving it through the net. Some examples of script-nets are presented in [3,4].

We allow multilabeling of net [5], i.e., labeling where each transition may be labeled by a set of expressions, such as sending and receiving a message, or creating a new object. This extension can simplify specifications and make them more compact.

By collecting communicating script-nets, we can produce a script system that represents a call center's logical structure. In this structure we can distinguish application scripts and system scripts representing system services such as resource management and call routing.

3.2 Macronets: exception handling

At this point, we should note that scripts describing real scenarios are usually extremely complicated to work with and therefore require some means of modularization. We will consider the problem of structural representation of script nets. In this respect, we can point out two techniques for modularization in Petri net-based models we would like to employ – hierarchical transitions and macroplaces. The first technique is well elaborated within the framework of high-level Petri nets, e.g. see [8]. Generally, it consists of representing a hierarchical net as a set of disjoint subnets with links between transitions and subnets forming a hierarchical structure. Firing of such a hierarchical transition causes an execution of its internal net that consists of the firing of a transition (or step) sequence from initial marking to the terminal one. So using this technique we can represent script nets as a set of hierarchical organized script subnets.

At the same time, in call processing, we may face situations, which are asynchronous to normal processing, and a reaction to such events should also be specified. For example, there may be situations when, during the dialogue between agent and client, the telephone line is disconnected (e.g. suddenly client puts down a receiver); as well as more sophisticated situations when the processing of current calls is interrupted and the agent is forwarded to process new calls with higher priority. Moreover, processing of such broken calls could be recommenced upon availability of agents. To specify such situations in script nets, special constructs are needed. To accomplish this, we suggest using the concept of macronets reported in [2] and generalized on high level Petri nets [4].

Petri nets with macroplaces. Notions of macronets and macroplaces have been introduced in [2] for specification of such situations where starting the execution of one procedure may interrupt execution of another procedure. Syntactically, a macronet is defined similar to nets with hierarchical transitions, however we use macroplaces instead of transitions. In other words, a macronet could be perceived as a set of Petri nets equipped with hierarchical links of the type "*place $\rightarrow$ net*".

Graphically, a macronet can be represented as a set of included nets, each internal net being drawn within a circle of corresponding macroplaces. The head place of an internal net is marked by an incoming extra arc.

The firing rules of macronets are as follows:

- A macroplace is considered to have a token if its internal net also has a token;

- adding a token to a macroplace results in adding a token to the head place of the internal net;

- removing a token from macroplace results in removing a token from the internal net no matter what position it is in.

It is clear that the concept of a macroplace is helpful for representing various situations where an interruption is involved.

High level macronets. Using standard possibilities of high-level nets we generalize the notion of macroplaces, allowing us to specify more general constructions. First, we will be able to build constructions where execution of an internal net can be interrupted only in specified regions. Second, we can specify a head place of internal nets dynamically, helping us to inject a token into any desired place.

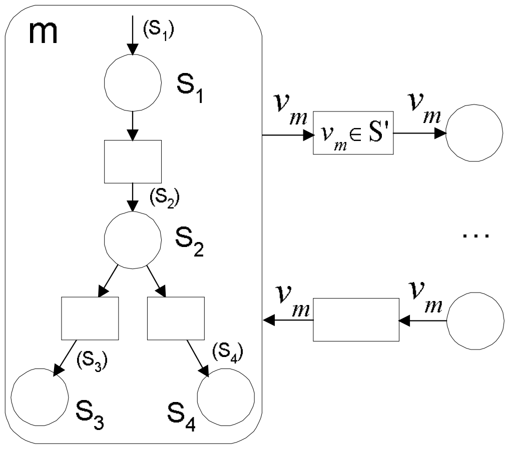

Figure 2: Macroplace and internal net

Let m be a macroplace and $N_m=(S_m,T_m,F_m)$ be its internal subnet. For an internal net we introduce a data type $type_m$ with a domain equal to a set of internal places: $Dom(type_m)=S_m=\{s_1,\dots,s_n\}$. Let us add to the token internal net an item of type $type_m$, the value of the item is exactly equal to the place where the token is situated. This can be easily implemented by assigning to tuples of incoming arcs with corresponding values from S_m, see Figure 2. Then we add to this a precondition of outgoing transition $t\in m^\bullet$ an expression of the type $v_m\in S'$, where $S'\subseteq S_m$. Firing of the transition t results in removing a token from a place of S'. Note that we can 'remember' the actual place where the token was before it had been removed. For an arc coming in to the macroplace m, the corresponding item of the tuple is stated in a suitable manner. For instance, if it is equal to a place $s_i\in S_m$ the adding of a token to macroplace m will result in injecting a token into s_i of the internal net. Apart from a constant, we can write a variable of the type $type_m$ that allows us to determine the incoming place dynamically. Specifically, we can replace taken to its point of origination, see Figure 2. More strict definition of high-level macronets can be found in [3].

With the aid of a macroplace, one can easily specify the next situation in an agent's scenario:

- Interruption of a script execution (naturally with an apology to a client) at any stage with subsequent return to an initial state. In this case the processing of the interrupted call is cancelled

- Interruption of a script execution only if it is in special regions of the script.

- Interruption of a script execution while noting the place of interruption and possible current parameters of the call processing. This information can be used for future recommencing of the processing of the call.

3.3 Time constraints

To represent real-time constraints that are very critical for scripts, we incorporate Merlin's time constructs [9] into our model. In particular, each transition in a script-net can be associated with a pair $[t_{min},t_{max}]$ that provides a time interval enabling the transition. This enables us to specify timeouts in script execution.

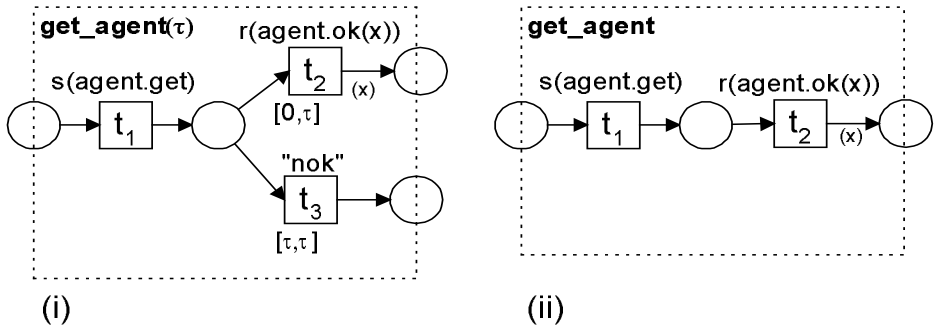

Figure 3: Resource capturing

Using this client server scheme of communication and time constraints, we can build a mechanism of resource capturing/release. On Figure 3 (i) the scheme of agent capturing within time interval τ is depicted. The capturing is started by sending c command ***get*** (firing of the transition t_1) to the script ***agent*** that defines the behavior of the agents. If within the time interval $[0,\tau]$ a positive reply is received ***r(ok(x))*** (firing t_2) then the agent with that identifier ***x*** is considered to be captured. If within time interval nothing happens then transition t_3 fires hence there are no agents available. This construction is called ***get_agent(τ)***. If no time interval is specified then the transition t_3 will never fire and thus can be removed, see Figure 3 (ii).

4. Examples

In the section we discuss a methodology of representing scripts according to the model we are introducing, with the aid of some realistic script examples.

Example 1. On Fig.4 the script corresponding to resource of operators (agents) is depicted. The agents can be in three states: READY, BUSY, NOT-READY (NR for short). The transition from READY to BUSY is caused by receiving a command ***get*** from an object ***X*** and sending it a reply ***ok***. Note that this transition is labeled by two labels that correspond to receiving and sending commands. In a BUSY state the agent can return to a READY state by receiving a command ***r(X.free)*** from the application. Moreover, in a BUSY state the agent can move to a NOT-READY state (e.g. switching to more urgent work) informing the application by sending a command ***not_ready***. The place S says that the operator a can work with script A, the operator b can work with script B and c can work with both scripts A and B.

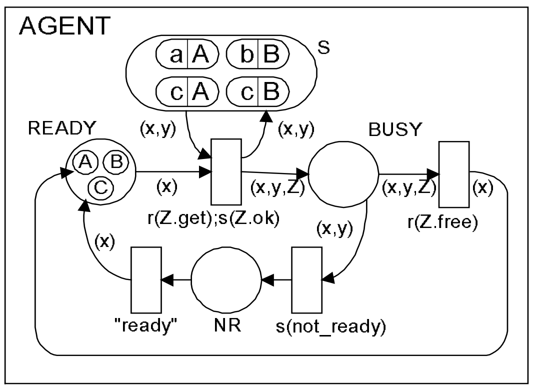

Figure 4: Script-net of agent.

Example 2. In this example we build a script-net corresponding to a script for the processing of a retail catalogue sales call center [9]. In this context, many customers call to inquire about the availability of items in a catalogue, status of their order, delivery options, and similar routine questions. Such simple calls can be processed automatically by a voice processing system. Other calls, especially those involving new customers who need special assistance, must be processed by an operator. In Figure 5, we present a script net corresponding to processing this type of call. When the system enters a call into a script (a token appears in a head place s_0), it plays a greeting and gives a choice of pressing "1", "2", or "3" (the transition t_1). These choices correspond to calls concerning availability of items, the status of a current order, or other types of calls, respectively.

In the first two cases the call is processed automatically (hierarchical transitions t_2 and t_4). The third case needs the intervention of an operator, who is captured by the expression ***s(agent.get).*** Here, the agent is the name of the script-net for resources corresponding to operators, get the name of capturing command. If there is a free operator in the system, he is captured. At this point the call is transferred and the operator works with the customer (hierarchical transition t_{10}). At the end of the conversation, the operator is released *(t_{11})*. If there are no free operators *(t_{12})* the system plays a recorded sound file with appropriate explanations**.**

In this example, two possibilities for agent capturing within the time interval τ are shown within dashed boxes. The capturing is initiated by sending a ***get*** command to the "agent" script that defines the behavior of the agents. If a positive reply ***r(ok)*** is received within the time interval represented by $[0,\tau]$ then the agent with that identifier is considered to be captured. If within the time interval nothing happens, then transition ***nok*** fires, indicating that no agents are available. This construction is called ***get_agent(τ).***

Imagine that an agent involved in call processing presses a "not ready" button on his telephone and becomes unavailable. This event corresponds to firing the transition t_{14}. At this point the script tries to find another agent (t_{15}). If another agent is indeed available, control of the script is returned to the same place where it was interrupted, and call processing continues. The state where the call processing was interrupted is saved in the variable v.

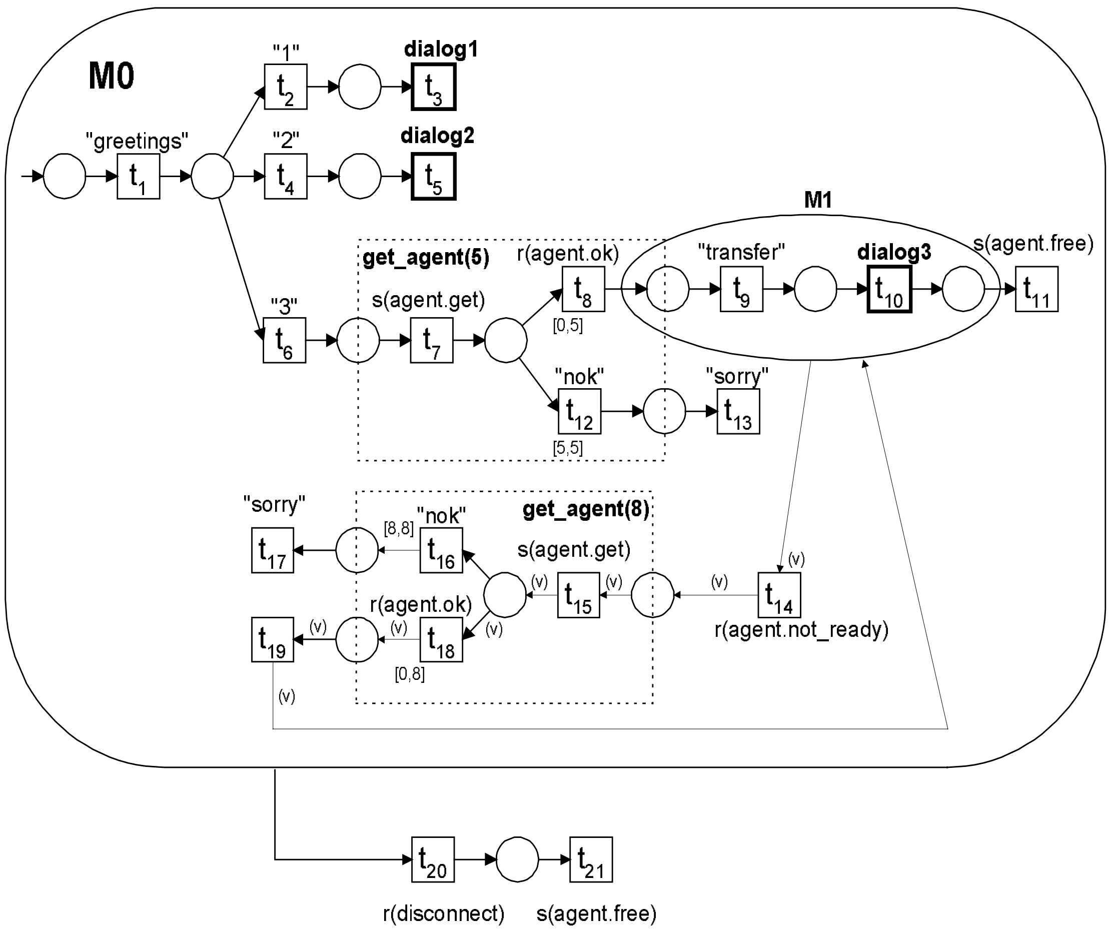

Figure 5: Example of a Script-Net, in Catalogue Sales Context

Alternatively, imagine that a client suddenly hangs up during a call. This event corresponds to firing the transition t_{20}. Firing of this transition disrupts the execution of the script including the construction defined above, releases its active agent (if any), and then terminates the call processing.

4. Implementation Issues

In this section we briefly sketch some tools for generation call center applications based on developed formal model. In particular, we developed a programming system for building scripts for processing both inbound and outbound calls with extensive intervention of living agents. The system functions in an Intranet environment, is based on thin-client technology, and uses a graphic language to describe agent work scripts. More specifically, the system comprises the following components:

- A front-end graphical language for specification scripts with semantics based on Petri net-based model;
- A graphic script editor that supports the scripting language and allows to build scripts in a simple and convenient way;
- Form manager for creating a set of forms to be interchanged between application and agent station during a call processing;
- A script engine that executes scripts upon emerging inbound and outbound calls in the run-time stage.

The graphical scripting language allows one to represent a script as a graph where each node depicted by icon corresponds to elementary communication with other objects (e.g. devise object, agent). Arrows between icons define a causal relation between communication actions.

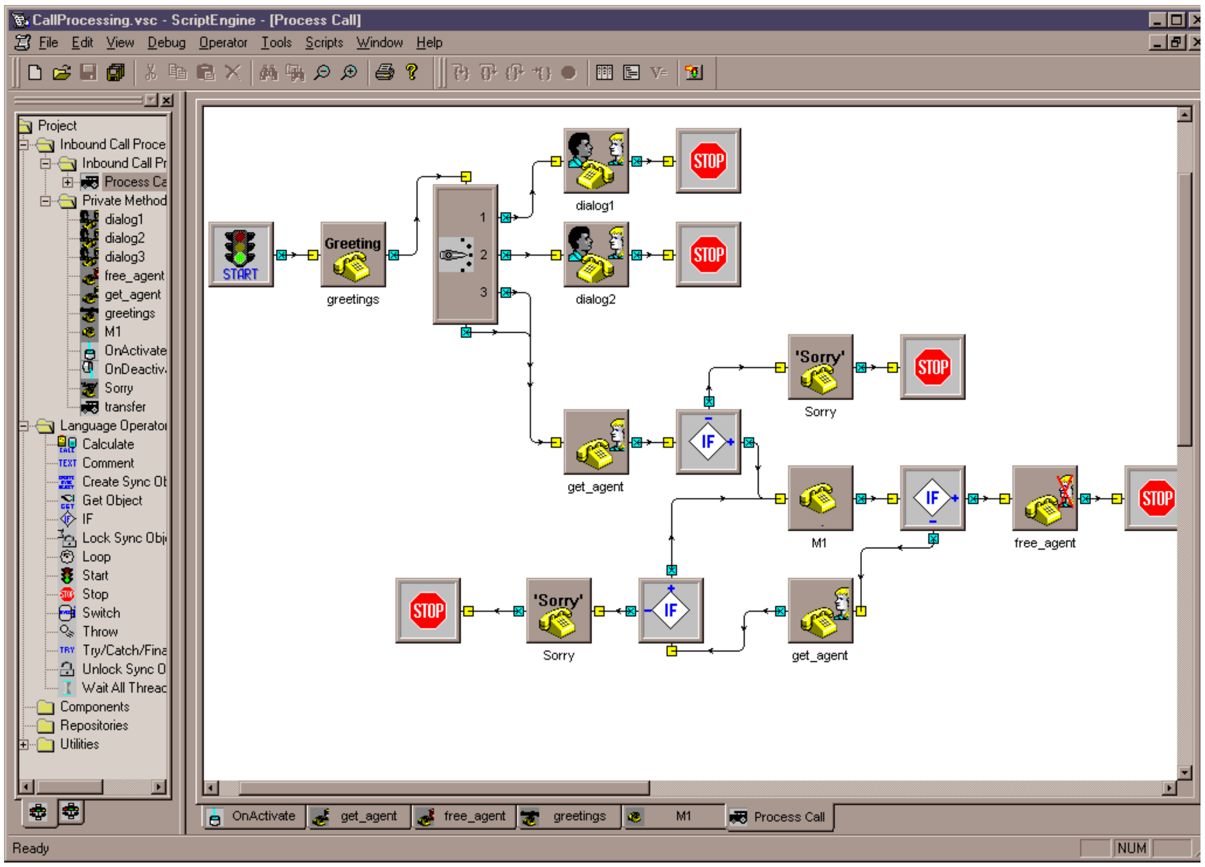

Figure 6: Graphical script editor

Among other features of the language, we can mention features which inherited from the formal model:

- Parallel constructs allowing one to represent multithreading and synchronization between threads;
- Hierarchical constructs which allow one to build scripts in a modular fashion;
- Exception handling constructs which enable one to specify the reaction of the scripts on receiving asynchronous and unsolicited events.

On Figure 6, we present the snapshot of the script editor with the fragment of the script discussed in the example 2.

5. Concluding Remarks

In this paper we have proposed the Petri net-based model for design different distributed CTI applications. This approach enables a designer to represent a logical structure of complex applications for call centers.

In the nearest future we plan to pay more attention to architectural aspects in the process of formalization taking into consideration different related architectural approaches [7].

Considering almost all CTI-applications are real time systems, we must take into consideration time and stochastic aspects of the model under discussion, i.e., it is desirable to calculate availability of a call center (for specific sort of calls), agent's loading, optimal configuration of call-center, etc. By extending our model towards stochastic Petri nets, these issues can be addressed.

Acknowledgements. The authors are grateful to Evgeny Petrovikh and Pavel Postupalski for they helpful discussions. First author had also a benefit from Russian Fund for Basic Research (grant No. 96-01-001773).

REFERENCES

[1] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. To appear in the Journal of Circuits, Systems and Computers (1998).

2. N.A.Anisimov. An Algebra of Regular Macronets for Formal Specification of Communication Protocols. Computers and Artificial Intelligence, Vol.10, No.1 (1991), pp. 541–560.

3. N.Anisimov, K.Kishinski, A.Miloslavski. Petri Net Based Model for Design of CTI-applications, in Proc. of the Int. Conference "Computational Engineering on Systems Application: CESA'96", July 9-12, 1996, Lille, France.

4. N.A.Anisimov, K.P.Kishinski, A.Miloslavski, P.A. Posupalski, Macroplases in High Level Petri Nets: Application for Design Inbound Call Center, In: Proc. Int. Conference on Information Systems Analysis and Synthesis (ISAS'96), Orlando, Florida, USA (July 22-26, 1996), pp.153-160.

5. N. Anisimov, M. Koutny. On Compositionality and Petri Nets in Protocol Engineering. In: Protocol Specification, Testing and Verification, XV. Chapman & Hall, pp.71-86, 1996.

6. C.A. Ellis and G.J. Nutt. Modeling and Enactment of Workflow Systems. In: M.Ajmone Marsan (ed.), 14th International Conference on Application and Theory of Petri Nets 1993, Lecture Notes in Computer Science, Vol. 691, pp.1-16. Springer-Verlag, Berlin, 1993.

7. Enterprise Computer Telephony Forum, S.100 Revision, Media Services "C" Language, Application Programming Interface, 1996.

8. K.Jensen, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1: Basic Concepts, EATCS Monograph on Theoretical Computer Science, Springer Verlag, 1992.

9. E. Margulies. Voice Processing Applications Flatiron Publishing, 1995, ISBN 0-936648-70-8

10. P.M.Merlin, D.J.Farber. Recoverability of Communication Protocols — Implication of a Theoretical Study. IEEE Trans. Commun. COM–24 (1976) 1036–1043.

11. J.L.Peterson. Petri Net Theory and the Modeling of Systems, (Prentice–Hall Inc., 1981)

12. W.Reisig. *Petri Nets: An Introduction.* EATCS Monograph on Theoretical Computer Science (Springer–Verlag, 1985).

13. C. Sibertin-Blanc, A Client-Server Protocol for the Composition of Petri Nets. In: M.Ajmone Marsan (ed.), 14th International Conference on Application and Theory of Petri Nets 1993, Lecture Notes in Computer Science, Vol. 691 (1993) pp. 377-396

14. C. Sibertin-Blanc, Cooperative Nets, In: R. Valette (ed.) 15th International Conference on Application and Theory of Petri Nets 1994. Lecture Notes in Computer Science, Vol.815, Springer Verlag (1994), pp.471-490

15. R.Walters. Computer Telephone Integration, Artech House (1993).