An Object-Oriented Simulation Program Generator

Nikolaos S. Papaspyrou (nickie@softlab.ntua.gr)

Aris N. Tsois (atsois@softlab.ntua.gr)

National Technical University of Athens
Department of Electrical and Computer Engineering
Division of Computer Science
Polytechnioupoli, 157 73 Zografou, Athens, Greece

Introduction

This paper presents the main ideas and preliminary results of the two authors' doctoral research, which is currently in a rather early stage. Up to this point, the work of the two authors is closely related and can be thought of as part of a project named "ARGUS", which is currently in progress at the Software Laboratory of the National Technical University of Athens. It is expected that the courses of the authors' research will divide in the future and that the results of this research will support two doctoral theses. The ARGUS project is funded by the NTUA and is directed towards research in the field of object-oriented simulation program generators. The research team consists of the two authors, who are currently Ph.D. candidates, and is supervised by professor Emmanuel Skordalakis.

An overview of simulation support tools

Various tools have been developed as an aid in the task of developing simulation programs. These tools can be classified in three major categories, with respect to the underlying concepts, modelling capabilities, ease of program development and consequently with respect to their target groups: "programming", "automatic programming" and "non-programming". Overall, simulation support tools have followed the evolution that is shown in Fig.1.

Tools belonging in the "programming" category include general purpose programming languages as well as specific simulation-oriented programming languages. They offer great modelling capabilities and efficiency; however they require advanced programming expertise and the development of complex simulation programs is time-consuming and prone to programming errors. Various "non-programming" simulation packages have also been developed, using description languages and requiring significantly less programming expertise and effort. Nevertheless, their capabilities are limited: they sometimes prove to be either too general or too specialized to be used efficiently.

The emergence of object-oriented programming languages has brought about a new approach towards simulation. The object-oriented programming paradigm is particularly appropriate for the development of large simulation programs [Thom90] [Eldr90]. Objects provide a very natural way of representing the components of a real system and features

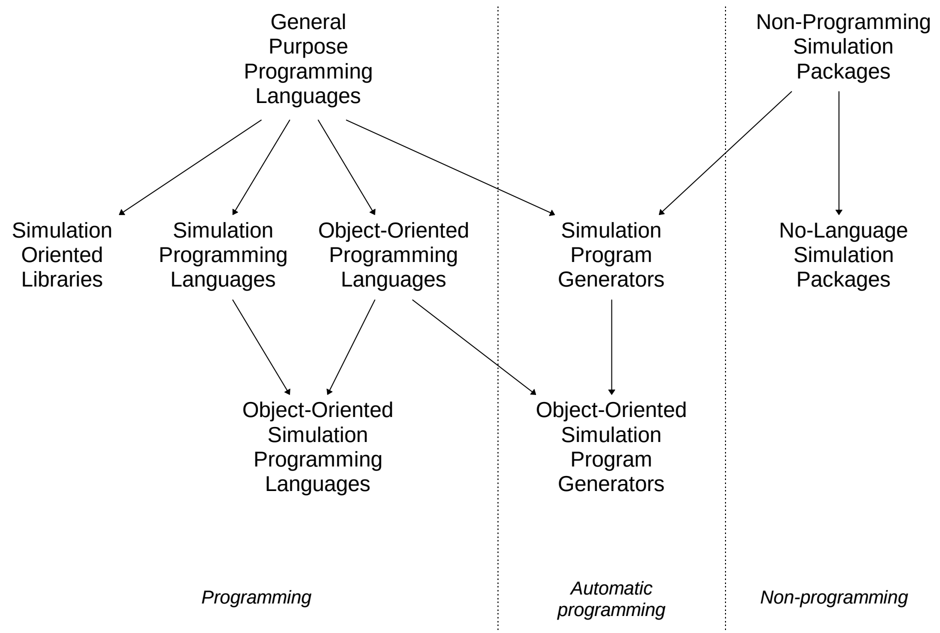
General Purpose Programming Languages
Non-Programming Simulation Packages
Simulation Oriented Libraries
Simulation Programming Languages
Object-Oriented Programming Languages
Simulation Program Generators
No-Language Simulation Packages
Object-Oriented Simulation Programming Languages
Object-Oriented Simulation Program Generators
Programming
Automatic programming
Non-programming

Fig. 1. Evolution of support tools for developing simulation programs.

such as encapsulation, inheritance and polymorphism can be extremely useful for the simulation of complex systems. Encapsulation allows the programmer to think of objects as black boxes, with properties and behaviour of their own. Inheritance and polymorphism can be used to define a hierarchy of objects, representing the components of the real system. In this hierarchy, objects can be thought of as specialized forms of other objects, and therefore they can share common properties and behaviour. Furthermore, one must not overlook that (well designed) objects are almost directly reusable.

Program generators and the "automatic programming" approach have also been suggested as a solution to the problem of writing error-free simulation programs without much effort. The construction of such tools is possible because of the similarities that the majority of simulation programs present. Simulation program generators generally receive as input a description of the real system in an appropriate description language and generate a simulation program in a programming language. This program can be later modified (if necessary), compiled and executed. Significant research has been conducted in the field of simulation program generators, resulting in valuable products such as SmartSim [Ulge90], QMG [Racz90], CAPS/ECSL [Clem82] and PASSIM [Shea90].

Current research in the field of simulation program generators is directed mainly towards two targets: finding a more natural way of description and taking advantage of object-oriented concepts. Following the tendency towards user-friendly, easy-to-learn and easy-to-use software, simulation support tools have been recently developed that are based on graphical user interfaces, visual programming and programming by demonstration [Smit94]. On the other hand, a new category of object-oriented simulation program generators is recently emerging. SmartSim [Ulge90], SmarterSim [Ulge89], using Smalltalk-80, and GASPE [Simo89] belong in this category.

The ARGUS project and its results so far

NTUA's research project ARGUS aims at improving the development of simulation programs by focusing mainly on: (1) the use of simulation program generators; (2) the application of the object-oriented paradigm in all phases of simulation development; (3) the use of visual tools for the efficient description and specification of real systems; and (4) the combination of all these and their application in large-scale simulation projects, where efficiency, abstraction, modularization and reusability are very important issues.

The ARGUS project so far has resulted in the development of an experimental tool with the same name.[1] This tool is a generator of simulation programs in C++, which targets mostly the area of queueing systems and combines the advantages of object-oriented simulation program generators with the convenience of a graphical user interface. In the beginning of the project, it was expected that experience gained through the specification, design and development of such a tool would be valuable for the progress of our research.

Our intention was to make the process of developing a simulation program as easy as editing. The description of the system to be simulated is given by means of a user-friendly graphical editor. As an alternative or a complement to the graphical editor, a description language named SCGL (Simulation Code Generation Language) is used. The current experimental system, whose complete block diagram is shown in Fig.2, supports three levels of operation, depending on the user's programming experience and the demands of the real system to be simulated. The trade-off between user-friendliness and flexibility is inevitable: by favouring one of these properties, we lose some of the other. A combination of all three levels is possible.

- On the first level, the user describes the system graphically by means of the graphical editor and no language (programming or description) is required at all.
- On the second level, the object-oriented description language SCGL is used. This approach is not as simple as the graphical description, but enables the user to describe more complex real systems.
- On the third level, the user can include C++ code in a real system's SCGL description. This code will be included in the simulation program. Although the third level provides the highest flexibility, it requires advanced programming abilities, a knowledge of C++ and of ARGUS intrinsics.

ARGUS exhibits object-oriented characteristics both in the way that a system's description is given and in the simulation programs that it generates. When describing a system, simulation objects can form a hierarchy in which properties and behaviour can be inherited. Both the graphical editor and the description language SCGL allow the user to define inheritance in a natural and effortless way. Furthermore, simulation programs created by ARGUS are written in C++ and make use of this language's object-oriented features to directly depict the hierarchy of objects.

[1] This tool is outlined in a paper entitled "ARGUS: A program generator for discrete-event simulation problems", which has been submitted for publication in the International Journal of Modelling and Simulation.

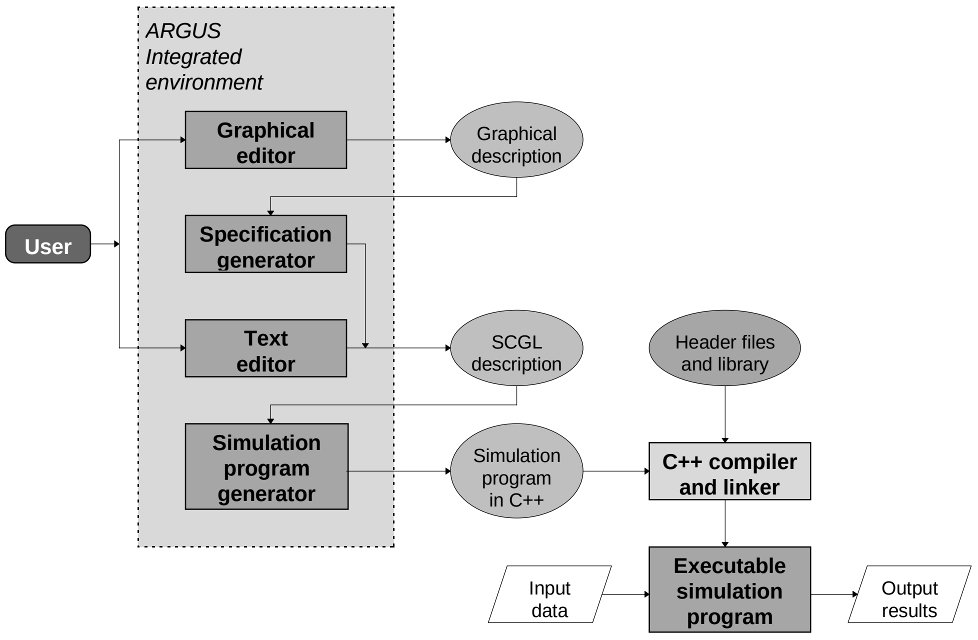

Fig. 2. Block diagram of ARGUS.

The suggested modelling process

In the current experimental system, the queueing system models that can be simulated by ARGUS consist of two kinds of objects: stations and entities. A station acts as a server in the queueing system, whereas an entity acts as a client. Stations determine the topology of the queueing system and entities represent its population. The life cycle of an entity is an iteration of the following steps:

1. The entity arrives at a station and is put in the station’s queue.
2. While being in a station’s queue, an entity can be in one of two possible states: “waiting” or “being served”. The transition between these two states is controlled by the station.
3. Eventually, the entity’s service is completed. This time is also determined by the station. The entity is then removed from the station’s queue and leaves the station.

Stations are usually the most important part of a system’s definition. Modellers must specify a number of characteristics for each station, such as: the type of the station, which determines the mechanism that is used for serving the incoming entities; the type of the station’s queue, which determines its length and what will happen if an incoming entity finds it full; the number of servers; the service policy, which determines typically how long a specific entity will have to remain at the station, and; the transition policy, which determines where entities will be send after their service is completed.

Both stations and entities are viewed as objects and multiple inheritance of station or entity characteristics is supported. Stations and entities can belong to hierarchies, where a station or entity inherits the characteristics of its parents. Furthermore, characteristics of derived stations and entities can be redefined. The users can also specify the simulation program’s output, that is the statistics about the real system that need to be collected.

ARGUS is meant to be a discrete event simulation program generator. Behind the scene, all transitions of entities as well as all changes in the state of entities occur as the results of simulation events. The simulation clock proceeds non-continuously from one event to the next. The most important type of events is the completion event, which signals the end of a period of service for a particular entity. As a result of a completion event, the entity may be put again in the station’s queue, may be sent to some other station or out of the system, according to the station’s transition policy.

Stations and entities are implemented in the generated simulation program as C++ classes and the whole runtime environment has been written in C++. The way in which C++ code is embedded in graphical or textual descriptions presents an important advantage. It is not necessary to alter the simulation program that ARGUS generates in order to implement something that ARGUS is not prepared for. By placing the code in the model’s description, modellers are free to edit the code or the description any number of times and then use ARGUS to generate a new simulation program. Emphasis has been given to the issues of readability and reusability of the generated code.

Critique so far

There are two major points in our critique of the ARGUS project so far. The first one is whether the expressiveness of our simulation program generator is adequate for real-world problems. We believe that there is a lot to be expected from the graphical editor, regarding its expressiveness. Part of the research is also dedicated to the task of making the graphical editor more intelligent. The description language SCGL should also be extended, in order to include features useful in the specification of large systems. The most important of the features that we intend to add to both graphical and textual descriptions is that of a subsystem. Subsystems will be collections of stations and other subsystems that, for reasons of organization and abstraction, can be thought as self-contained or independent.

The second point concerns the effectiveness of visual descriptions. Although the process of describing a system graphically is easy and relatively straightforward, there is a major drawback: the trade-off between user-friendliness on the one side and flexibility and efficiency on the other. Although using the graphical editor is easy for any modeller with a basic computer education, it may not be the first choice for a modeller with experience in description or programming languages. Research has shown that, when the size of the model grows, experienced modellers tend to prefer textual descriptions to a graphical ones. The main advantages of the former are: flexibility, efficiency, better organization and abstraction. Even though we tried to keep these in mind while developing the graphical editor and to emphasize on flexibility, efficiency and abstraction, we are not yet satisfied by the results.

Taking these two points into consideration, our future research will be directed towards: (1) improvement of the graphical editor and search for methods of making it more efficient; and (2) redesign of the description language and improvement of its abstraction and modularization capabilities. Also, on a different path, future research will aim at (1) improving the readability and reusability of the generated C++ code; and (2) improving the efficiency of the generated simulation programs, by generating parallelizable C++ code. Both these issues are very important for large-scale simulation projects.

Conclusion

The results of the ARGUS project so far are encouraging beyond doubt. Our main goal was to create an experimental simulation tool that is easy to use, without sacrificing its flexibility or generality. We believe that this is partially achieved by supporting three levels of operation in our experimental program generator, aiming at different categories of users, from modellers with no or little computer experience to programming experts. Also, we consider the use of a graphical editor and the use of object-oriented analysis and design techniques as fundamental characteristics of the simulation support tools that will prevail in the future. Although we have not yet reached our goal, we are optimistic about the project's outcome. Future research in the areas discussed earlier is expected to add to the project's maturity.

References

[Clem82] Clementson A. T., "Extended Control and Simulation Language", Cle. Com. Ltd., Birmingham, England, 1982.

[Eldr90] Eldredge D. L., McGregor J. D. and Summers M. K., "Applying the Object-Oriented Paradigm to Discrete Event Simulations Using the C++ Language", Simulation 54, pp. 83-91, 1990.

[Racz90] Raczynski S., "Graphical Description and a Program Generator for Queuing Models", Simulation 55(3), pp. 147-152, 1990.

[Simo89] Simonot F., LeDoeuff R., Haddad S. and Ramaromisa V., "An Object-Oriented System for the Automatic Generation of Simulation Programs in Power Electronics", CAD & CG '89 Beijing, Proceedings of the International Conference on Computer-Aided Design and Computer Graphics, pp. 807-811, 1989.

[Shea90] Shearn D. C. S, "PASSIM: A Pascal Discrete Event Simulation Program Generator", Simulation 55(1), pp. 31-38, 1990.

[Smit94] Smith D. C., Cypher A. and Spohrer J. C., "KidSim: Programming Agents without a Programming Language", Communications of the ACM, vol. 37, no. 7, pp. 55-67, 1994.

[Thom90] Thomasma T. and Madsen J., "Object Oriented Programming Languages for Developing Simulation-Related Software", Proceedings of the 1990 Winter Simulation Conference, pp. 482-485, 1990.

[Ulge89] Ulgen O. M., Thomasma T. and Mao Y., "Object Oriented Toolkits for Simulation Program Generators", 1989 Winter Simulation Conference Proceedings, pp. 593-600, 1989.

[Ulge90] Ulgen O. M. and Thomasma T., "SmartSim: An Object Oriented Simulation Program for Manufacturing Systems", International Journal of Production Research, vol. 28, no. 9, pp. 1713-1730, 1990.