AN OBJECT-ORIENTED SIMULATION PROGRAM GENERATOR

Nikolaos S. Papaspyrou, Aris N. Tsois and Emmanuel S. Skordalakis

National Technical University of Athens, Department of Electrical and Computer Engineering,
Division of Computer Science, Polytechnioupoli, 15780 Zografou, Athens, Greece.
Tel: +30-1-7722486, Fax: +30-1-7722519, E-mail: {nickie,atsois,skordala}@softlab.ntua.gr

Abstract

This paper traces the evolution of simulation support tools and outlines ARGUS, a discrete event simulation program generator that targets mainly the area of queueing systems. ARGUS supports three levels of operation, depending on the user's needs and programming expertise. The system to be simulated can be specified using an integrated graphical editor or a textual description language. Object-oriented concepts such as encapsulation, inheritance and polymorphism are employed in both the specification process and the generated simulation programs are written in C++. Throughout the paper, the main characteristics of ARGUS are summarized and its capabilities are briefly presented.

Keywords
Discrete event simulation, object-oriented simulation, simulation program generators, graphical editors, queueing systems.

1 Introduction

During the last decades, simulation has become a very popular method of analyzing and designing real world systems. Digital computers have helped in this, as the devices that automate the simulation process. The development of the required programs is not a trivial task and years of research have been spent in order to simplify it.

Program generators have been suggested as a means of developing simulation programs in an efficient and effortless way. In this paper, we describe ARGUS, an object-oriented program generator for discrete event simulation problems. ARGUS allows the modeller to use a graphical or textual description language for describing the simulation models and produces efficient C++ programs. The structure of the paper is as follows. In section 2 we present a categorization of existing simulation support tools. Sections 3, 4 and 5 describe the main characteristics of ARGUS. Finally, in section 6 we conclude with a discussion on the current status of ARGUS and suggest directions for future work.

© 1998 IEE, WODES98 – Cagliari, Italy
Proc. of the Fourth Workshop on Discrete Event Systems

2 Simulation support tools

Various tools have been developed as an aid in the task of developing simulation programs. These tools can be classified in several categories with respect to the underlying concepts, modelling capabilities, ease of program development and consequently with respect to the group of target users. They can be principally classified in three primary categories, representing three different approaches to simulation: the *programming* approach, the *automatic programming* approach and the *non-programming* approach. Overall, simulation support tools have followed the evolution that is shown in Fig.1.

General purpose programming languages (GPPLs) were the first tools to be used for the development of simulation programs. Their main advantages are the great modelling capabilities that they offer and the efficiency of the produced simulation programs. Unfortunately, in most cases their use presents many disadvantages, the most important being the need for experienced programmers. Furthermore, the development of complex simulation programs with GPPLs is time-consuming and prone to programming errors. The drawbacks in using GPPLs for complex simulation programs have led to the development of two new categories of tools. Both categories attempt to extend GPPLs by introducing characteristics useful to simulation developers.

The first approach, based on code reusability, enriches popular GPPLs with simulation-oriented libraries (SOLs). Such libraries provide basic features used frequently in simulation programs, such as lists, events, random number generators and collection of statistics. Characteristic examples of this category are SIMTOOLS (Seila, 1988) and LIBSIM (Crookes, Balmer, Chew and Paul, 1986), both for the Pascal programming language, and also SIMEX (University of Minnesota, MN, 1994) and C++SIM (Little and McCue, 1994) for C++. The second approach involves the development of simulation programming languages (SPLs). These languages are generally based on popular GPPLs with extensions to provide built-in basic simulation-oriented facilities. Among them, the most popular are SIMULA 67 (Nygaard, 1986), GPSS (Gordon, 1974), SIMAN (Ped-

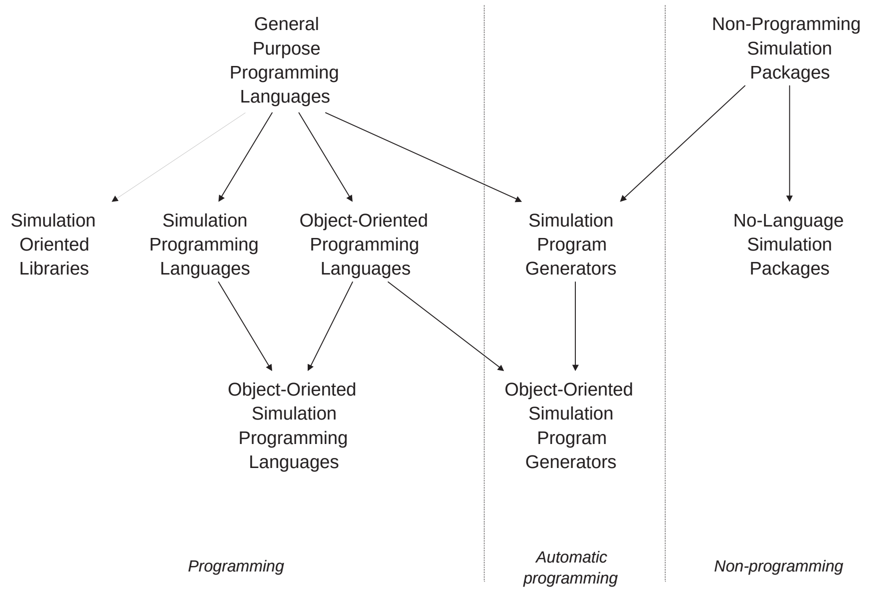

General Purpose Programming Languages
Non-Programming Simulation Packages
Simulation Oriented Libraries
Simulation Programming Languages
Object-Oriented Programming Languages
Simulation Program Generators
No-Language Simulation Packages
Object-Oriented Simulation Programming Languages
Object-Oriented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

Figure 1: Evolution of support tools for developing simulation programs.

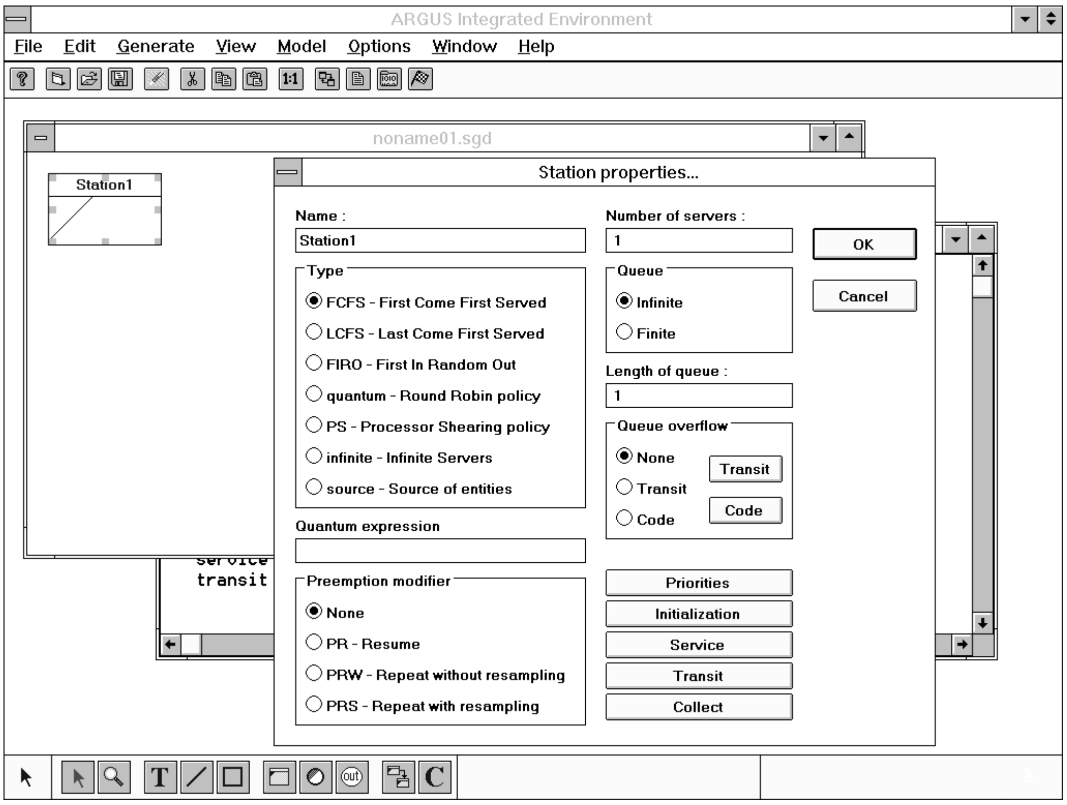
gen, 1987), SIMSCRIPT II.5 (Kiviat, Villanueva, Markowitz, 1975), and SLAM II (Pritsker, 1984) but several others should be noted, such as INSIGHT (SysTech Inc., IN, 1985), PCModel (White, 1988) and SIMPLE_1 (Sierra Simulations & Software, NH, 1989). The use of SOLs and SPLs substantially simplifies the programmer's job. However, knowledge of a programming language is still required and all associated problems, although less intense, remain.

Apart from programming languages, various *non-programming* simulation packages (NPSPs) have been developed, aiming at simulation developers with little or no programming experience. These tools range from general-purpose simulation packages to tools specialized for the needs of particular fields. In this category belong AutoMod II (AutoSimulations Inc., UT, 1989), ProModel (Production Modeling Corporation, UT, 1989), WITNESS (AT&T ISTEL, OH, 1989), SIMFACTORY II.5 (CACI Products Company, CA, 1990) and XCELL+ (Conway and Maxwell, 1987). Such simulation packages use description languages and require significantly less programming expertise and effort than programming languages. However, their capabilities are limited: they sometimes prove to be either too general, and therefore inadequate for the needs of a specific field, or too specialized to be used in a situation somewhat different from the anticipated.

The emergence of object-oriented programming languages (OOPLs) has brought about a new approach towards simulation. Object-oriented programming is particularly appropriate for the development of large simulation programs [Thom90, Wegn92, Eldr90]. Objects provide a very natural way of representing the components of a real system and features 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 attributes 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 can share common attributes and behaviour. Furthermore, one must not overlook that (well designed) objects are almost directly reusable, whereas code written in a procedural programming language seldom is. Object-oriented simulation programming languages (OOSPLs), such as MODSIM II (CACI Products Company, CA, 1989) and Sim++ (Jade Simulations International Corporation, Canada, 1989), have been developed recently, combining simulation-specific facilities with object-oriented concepts.

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 (SPGs) generally receive as input a description of the real system and generate a simulation program in a programming language (general-purpose or simulation-oriented). This program can be later compiled and executed, or even modified if necessary.

Several SPGs have been developed during the last twenty years. Some of them are mostly oriented towards a particular field of applications, e.g. SmartSim [Ulge90], QMG [Racz90] for manufactur-



Station properties...

Name : Station1

Number of servers : 1

Type: [✓ FCFS - First Come First Served] LCFS - Last Come First Served / FIRO - First In Random Out / quantum - Round Robin policy / PS - Processor Shearing policy / infinite - Infinite Servers / source - Source of entities

Queue: [✓ Infinite] Finite

Length of queue : 1

Queue overflow: [✓ None] Transit / Code

Quantum expression :

Preemption modifier: [✓ None] PR - Resume / PRW - Repeat without resampling / PRS - Repeat with resampling

Figure 2: Screen snapshot while working with ARGUS.

ing systems, AGVS-SCG [Gong90] for automated guided vehicle systems, AS/RS-SPG [Asha89] for automated storage and retrieval systems. Others are more general, e.g. CAPS/ECSL [Clem82], AUTOSIM [Paul87], SSIM/DRAFT/DRAW [Math90] and PASSIM [Shea90]. Current research in the field of SPGs is directed mainly towards two targets: finding a more natural way of describing the system to be simulated and taking advantage of object-oriented programming concepts.

The need for representing the real system in a more natural and intelligible way arises from the fact that SPGs are generally used by people with little or no programming experience. Following the tendency towards user-friendly, easy-to-learn and easy-to-use software, it seems that the most suitable way of achieving this is by means of graphical environments [Ozde91]. Simulation support tools have been recently developed that do not use a programming or description language but are based on principles such as graphical user interface, visual programming and programming by demonstration. KidSim [Smit94] and Playground [Fent89], although hardly adequate for real-world simulation problems, take advantage of such methodologies to make simulation programming easy enough to be practiced by children.

On the other hand, the advantages of using object-oriented concepts in the development of simulation programs have already been explained. The number of OOPLs, that are used in the development of simulation software, provides definite proof. A new category of object-oriented simulation program generators (OOSPGs) is recently emerging. SmartSim [Ulge90], SmarterSim [Ulge89], using Smalltalk-80, and GASPE [Simo89] belong in this category.

3 ARGUS

We have developed an experimental tool named ARGUS that combines the advantages of OOSPGs with the convenience of a graphical user interface. ARGUS targets mostly the area of queueing systems. Our intention was to make the process of developing a simulation program with ARGUS 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, ARGUS uses a description language named SCGL (Simulation Code Generation Language). Fig.2 is a snapshot taken from the computer screen while working with ARGUS.

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, objects can be organized in a hierarchy in which attributes and behaviour are inherited. Both the graphical editor and the description language SCGL allow the user to define inheri-

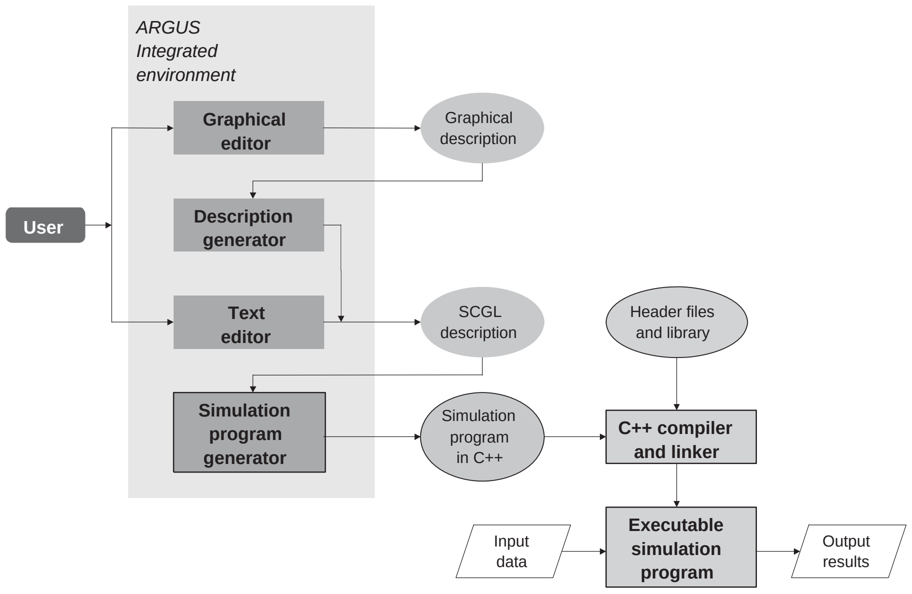


Figure 3: Block diagram of ARGUS.

tance 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 implement the hierarchy of objects.

The block diagram of ARGUS is shown in Fig.3, which illustrates the process of creating an executable simulation program. ARGUS itself consists of three parts: (i) the integrated environment, combining the graphical and text editors with the description generator, whose output is the system's description in SCGL; (ii) the program generator, which converts a description in SCGL into a simulation program in C++; and (iii) the run-time environment necessary for compiling the simulation program. A C++ compiler and linker are necessary to create an executable simulation program.

ARGUS 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 textual language is required at all. On the second level, the object-oriented description language SCGL is used in order to describe the real system. This approach is not as simple as the graphical description but it enables the user to describe more complex 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.

4 Queueing system models

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. The set of stations determines the system's topology and is static, while the set of entities represents its population and is dynamic. When entities arrive at a station, they are placed in the station's queue. Eventually they are served, according to the station's serving policy, and are sent to other stations.

The definition of stations is the most important part in the system's model. The station's *type* determines the serving mechanism for incoming entities. Predefined station types implement the most frequent serving mechanisms, such as FIFO and processor sharing. Stations can also serve as sources of entities in the queueing system. The characteristics of the station's *queue* (e.g. its length) can also be defined, as well as the number of entities that can be served at the same time. The *service* and *transit* properties determine the way in which an entity is served (typically it is only delayed for a random time that follows a given distribution) and the entity's destination after it is served. A station may also have user defined *attributes*.

ARGUS allows the modeller to define new types of stations or entities. Furthermore, it supports inheritance of station or entity characteristics, organizing them in hierarchies that have the structure of DAGs. A station or entity inherits the characteristics of its parents; however it has the option to override them. The user can also specify the simulation program's output, that is, what statistics about the real system are relevant and have to be collected. ARGUS supports directly the collection of statistics giving various important factors in simulation problems, such as response and waiting times, number of entities in queues and utilization ratios.

5 The three levels of operation

Level 1: Using ARGUS's graphical editor to describe a queueing system's model is a fairly simple task, requiring no more skills than using a word processor or drawing application. The modeller can use a set of tools for designing the system's stations and entities on the screen, using the mouse. Apart from stations and entities, decorative objects can be included in the system's description, such as rectangles, lines or descriptive text. In order to specify some properties of stations or entities, the modeller is presented with dialog boxes such as the one shown in Fig.2. The set of transitions within the system's stations can be specified by drawing directed lines between the stations on the screen.

Although the process of describing a system graphically is easy and relatively straightforward, there are a few drawbacks, the most significant being the trade-off between user-friendliness on the one side and flexibility and efficiency on the other. Research has shown that, when the size of the model grows, experienced modellers tend to prefer textual descriptions to graphical ones. Typical advantages of the former are flexibility, efficiency and organization. We believe that the lack of these qualities to some extent is indigenous in graphical environments.

Level 2: ARGUS's description language SCGL can be used for specifying models of complex queueing systems. As a description language, SCGL contains no algorithmic part. We tried to keep the language as simple as possible. There are no data types, except for types of stations and entities, nor control statements. Whenever a description of an algorithm is necessary, C++ code can be directly embedded in level 3 models. Nevertheless, the set of level two models that can be described without any knowledge of C++ is fairly large.

Consider the simple queueing system representing a small bank, which contains a station named *door*, with a FIFO queue. Suppose that the door's service time follows a uniform random distribution with values between 1 and 3 seconds. Suppose also that the entities leaving the door are directed to stations *desk1*, *desk2* and *desk3* with probabilities 0.5, 0.3 and 0.2 respectively. The SCGL description of the door is as simple as:

```
station door {
   type    : FIFO;
   service : delay uniform(1, 3);
   transit : desk1 with 0.5,
             desk2 with 0.3,
             desk3;
};
```

Consider now two additional hypotheses. First, the bank has two doors. Second, two types of entities arrive: normal clients and special clients. The latter always know which desk they want to go to. The description of the same part of the model now becomes:

```
typedef entity NormalClient;
typedef entity SpecialClient {
   inherits  : NormalClient;
   attribute : { station * dest; }
}

typedef station Door {
   type    : FIFO;
   service : delay uniform(1, 3);
   transit : case normalClient do
                desk1 with 0.5,
                desk2 with 0.3,
                desk3
             case specialClient do
                ENTITY(specialClient)->dest;
};

Door leftDoor, rightDoor;
```

In this description, a station type is defined for the door and two instances are created. Furthermore, two types of entities are defined. Special clients inherit all attributes and behaviour of normal clients, with the addition of attribute *dest* which determines their destination after they leave the door. The *transit* property of the doors distinguishes between the two types of clients and makes use of attribute *dest*.

Level 3: The lack of algorithmic part in SCGL is compensated by the use of C++ code, which the modeller can embed directly in critical parts of SCGL descriptions. This code will be included in the appropriate point of the simulation program. However, the modellers need to understand how the generated simulation program works and how their code will interfere with automatically generated code. By using C++ code, modellers can define new types of behaviour for stations and entities. They can also enforce the execution of procedures at specific points during the simulation, e.g. whenever a particular entity is created or served. Moreover, C++ code can be used to collect specific statistical information about the simulation that cannot be collected by means of the predefined SCGL statistics collection mechanism.

The way in which C++ code is embedded in SCGL descriptions (or even graphical 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.

6 Conclusion

The use of program generators for simulation has proved to be beneficial. Program generators combine the conciseness and facility of description languages with the flexibility and efficiency of programming languages. ARGUS is a discrete event simulation program generator that combines the advantages of a graphical editor with object-oriented characteristics. Models of real systems can be described by means of a graphical or textual description language and can contain embedded C++ code. Three levels of operation are supported, aiming at all categories of users, from modellers with no or little computer experience to programming experts.

Our main goal was to create an easy-to-use simulation tool without sacrificing its flexibility. Although we have not yet reached our goal, we believe that such a task is achievable and that ARGUS is a step closer to its accomplishment. Future research and work will focus mainly on three points: extending the description language SCGL, improving the graphical editor and investigating better ways of visualizing the object-oriented paradigm in simulation.

References

[Asha89] J. Ashayeri and L.F. Gelders, "Simulation program generator for AS/RS systems", in *Proceedings of the 10th International Conference on Automation in Warehousing*, pp. 209–220, 1989.

[Clem82] A.T. Clementson, *Extended control and simulation language*, Cle. Com. Ltd., Birmingham, England, 1982.

[Eldr90] D.L. Eldredge, J.D. McGregor and M.K. Summers, "Applying the object-oriented paradigm to discrete event simulations using the C++ language", *Simulation*, vol. 54, pp. 83–91, 1990.

[Fent89] J. Fenton and K. Beck, "Playground: an object-oriented simulation system with agent rules for children of all ages", in *Proceedings of OOPSLA '89 ACM*, pp. 123–137, 1989.

[Gong90] D.C. Gong and L.F. McGinnis, "An AGVS simulation code generator for manufacturing applications", in *Proceedings of the 1990 Winter Simulation Conference*, pp. 676–682, 1990.

[Math90] S.C. Mathewson, "Simulation modelling support via network based concepts", in *1990 Winter Simulation Conference Proceedings*, pp. 459–467, 1990.

[Ozde91] M.B. Ozden, "Graphical programming of simulation models in an object-oriented environment", *Simulation*, vol. 56, pp. 104–116, 1991.

[Paul87] R.J. Paul and S.T. Chew, "Simulation modelling using an interactive simulation program generator", *Journal of the Operational Research Society*, vol. 38, no. 8, pp. 735–752, 1987.

[Racz90] S. Raczynski, "Graphical description and a program generator for queuing models", *Simulation*, vol. 55, no. 3, pp. 147–152, 1990.

[Shea90] D.C.S. Shearn, "PASSIM: a Pascal discrete event simulation program generator", *Simulation*, vol. 55, no. 1, pp. 31–38, 1990.

[Simo89] F. Simonot, R. LeDoeuff, S. Haddad and V. Ramaromisa, "An object-oriented system for the automatic generation of simulation programs in power electronics", in *CAD&CG '89 Beijing, Proceedings of the International Conference on Computer-Aided Design and Computer Graphics*, pp. 807–811, 1989.

[Smit94] D.C. Smith, A. Cypher and J.C. Spohrer, "KidSim: programming agents without a programming language", *Communications of the ACM*, vol. 37, no. 7, pp. 55–67, 1994.

[Thom90] T. Thomasma and J. Madsen, "Object oriented programming languages for developing simulation-related software", in *Proceedings of the 1990 Winter Simulation Conference*, pp. 482–485, 1990.

[Ulge89] O.M. Ulgen, T. Thomasma and Y. Mao, "Object oriented toolkits for simulation program generators", in *1989 Winter Simulation Conference Proceedings*, pp. 593–600, 1989.

[Ulge90] O.M. Ulgen and T. Thomasma, "SmartSim: an object oriented simulation program for manufacturing systems", *International Journal of Production Research*, vol. 28, no. 9, pp. 1713–1730, 1990.

[Wegn92] P. Wegner, "Dimensions of object-oriented modeling", *Computer*, vol. 25, pp. 12–20, 1992.