

THE GPSS++ MODELLING LANGUAGE: OBJECT-ORIENTED VERSION OF GPSS

Danko Basch, Igor Capan, Tomislav Ivančić, Hrvoje Prgeša, Tomislav Sukser

University of Zagreb, Faculty of Electrical Engineering and Computing,
HR-10000 Zagreb, Unska 3, Croatia

danko.basch@fer.hr (Danko Basch)

Abstract

This paper describes the modelling language GPSS++. One of the main purposes of the language is its use in education. It extends the well-known discrete-event modelling language GPSS in several aspects, among which an object-oriented modelling is one of the most important. Other extensions of GPSS++ are: support for modular and hierarchical modelling, support for hybrid modelling (discrete and continuous), and significantly enhanced features for description of model behaviour (by using features of a general purpose programming languages). Syntax of the language is also modernized. In the same time, GPSS++ tries to retain the basic GPSS concept of modelling that uses transactions and blocks. The paper describes the main features of the GPSS++ mentioned above. Results of comparison of models written in GPSS and GPSS++ are given. The main advantages of GPSS++ over old versions of the language are better organization of the model and improved readability and writeability. Also, the language is better adapted to the programmers with the background in modern and widely used object-oriented languages. In addition, GPSS++ is better suited for larger models due to its modular and hierarchical organization. GPSS++ is currently under development and here we describe its first version and we also give directions for future improvements and research.

Keywords: Modelling languages, GPSS, Object-oriented languages.

Presenting Author's biography

Danko Basch received B.Sc. in electrical engineering (1991), M.Sc. in computer science (1994), and Ph.D. also in computer science (2000) from the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, Croatia. In 1992 he joined the FER (Department of Control and Computer Engineering) as a researcher. At present, he works at the same Department as an associate professor. His research interests include programming language design and implementation, and also modelling and simulation languages.



1 Introduction

GPSS [1] is one of the oldest simulation languages - its 40th anniversary was celebrated in 2001. However, it is still in use and under active development, as noted by panellists in [2]. GPSS has undergone many changes through many versions, but the most valuable legacy of GPSS, its concept, has been retained in all versions. The concept is very simple, natural, and straightforward, yet very powerful and flexible, and afterwards, it has been used not only in GPSS, but also in many other simulation languages and tools. The main idea is to let many transactions simultaneously flow through blocks that represent a model structure.

But, GPSS has come to age. Its syntax is partially freed from rigorous and old-fashioned assembler-like format in later versions (starting with GPSS V), but from today's point of view it is still arcane and strange to most of programmers. Some versions of GPSS are extended by hybrid modelling abilities (e.g. GPSS World [3]). Also, some tools allow graphical modelling (e.g. WebGPSS [4]). Many new blocks and features have been added through years, making GPSS more powerful, but also rather complex [5]. Previous suggestions for changes of GPSS are summarised in a paper by Ståhl [6]. Here we briefly repeat the most important ones. They are: simplification, addition of modularity, structural modelling, and designing a new version based on GPSS and C. Backward compatibility will certainly be lost in the case of syntax changes, but some authors also propose that the new language should be made from scratch [7].

When speaking about the improvements, it is necessary to mention object oriented (OO) paradigm (OOP). OOP is a concept invented in the modelling community in the 1960s, introduced in well-known languages Simula I and Simula 67 [8, 9]. It seems that, during the following years, objects and classes somehow slipped out from the major modelling languages. But, OOP was widely accepted in the programming community some 20 years ago, when a general purpose programming language - C++ was introduced [10]. Today, it is hard to imagine a new programming language that has no OO features. In the last ten years or so, OOP begins to return to its roots [11]. As examples, we can mention OO version of Simscript [12] and Simple++ [13]. General-purpose OO programming languages are also used in simulation, e.g. Java in SSJ [14].

Education was the main motive for the development of a new version of GPSS. One author is involved in teaching a modelling and simulation courses to undergraduate and graduate computer science and engineering students for several years. Most of the students are familiar with modern programming languages like C [15], C++ [10], Java [16], or C# [17], and they have no problem in grasping the main concepts of GPSS, but they found the language itself as the biggest obstacle in writing models.

As with any new language, one may ask is it really necessary. For each new language the answer could be "No" since everything could be programmed using assembly language. But, new languages bring new ideas that will influence future languages, even if the language itself never becomes widely used.

There are many modelling and simulation languages available on the market. We could mention two of them that are well known and in widespread use: Modelica and Matlab.

Our students use Matlab in another course (but only for continuous modelling). One of its drawbacks is its cost and students are unable to use it at home. Modelica is free, but it still has a main disadvantage that we also found in Matlab. Both simulations packages are very powerful, and both allow hybrid modelling, but they are much more appropriate for continuous modelling. Since we need a classical discrete event modelling, we found them significantly less appropriate than the languages that are designed primarily for discrete models, like for example GPSS, SLX, Simscript or Modsim. There is also a Matlab-toolbox that enables script writing in a GPSS-like language [18], but it basically offers the same features as other GPSS dialects.

The price is one of the main concerns, since we want our students to be able to work at home and, if possible, for free. Many simulators are available at reduced fees for students but with unacceptable evaluation periods, limitations in model size or simulation duration, etc.

Among existing discrete modelling languages, from the viewpoint of education, we found transaction-oriented modelling offered by GPSS more appropriate than event/process-oriented modelling, although it must be said that this is only a matter of personal preference. It is true that an event/process-oriented modelling offers a greater flexibility in some cases, but in our opinion it is more complicated, less problem-oriented, and it exposes more of the underlying simulation algorithm.

A year ago we have started the development of a new version of GPSS as a student project. The language was named GPSS++. The compiler for GPSS++ is finished, and the simulator is still under development. We expect it to be operational until the end of this year. We plan to use it for a year or two in education. After that, we plan to design and implement the second version. It should be designed and improved according to the collected experience and feedback from the students (and hopefully from a wider community).

The main features of GPSS are briefly described in the following section. The rest of the paper introduces the GPSS++ language, with explanation of the basic design principles followed by the main language features introduced in more details. We compared

several models written in old and new GPSS, and the comparison results are presented here as well. At the end of the paper we give the proposals for the future work and refinements of GPSS++.

2 A short introduction to GPSS

For a reader unfamiliar with the GPSS language, we shall briefly describe its main features. GPSS is a discrete-event modelling language that supports transaction-oriented modelling. Its main notions are transactions and blocks, and it also uses static simulation entities.

Dynamic entities in GPSS that enter and leave a model are called transactions (e.g. a transaction can represent a customer in a bank model).

Static entities, called facilities and storages in GPSS, represent the resources of a system. Transactions are served by the static entities. Facility can serve one transaction at the time, while storage can contain several transactions (e.g. facility can represent a teller, and storage can represent a group of tellers in a bank).

Blocks form sequences through which transaction passes during simulation. Blocks describe lifetime of a transaction, or structure of the system through which transaction passes. Blocks are used to define: creation of transactions, capturing and freeing of resources, duration of activities, collection of statistics, conditional movement of transactions, removal of transactions from the model, etc.

The sequences of connected blocks, called segments, are the main parts of the models in GPSS. They are preceded with declarations, and followed by statements for controlling the experiment.

We shall use a simple model of a barbershop to illustrate the features of GPSS. First, the system will be described, and then the model in GPSS will be given with a more detailed explanation.

Our system will be a barbershop where customers arrive with the inter-arrival time (IAT) distributed uniformly with a mean of 12 and spread of 6 minutes. The first customer will arrive at the first minute, and at most 400 customers will be generated during the simulation. 25% of customers need haircut, and 75% need shaving. There are two barbers working in the barbershop, and each of them has his own scissors but they have only one razor. Haircut takes from 12 to 24 minutes, and shaving from 8 to 12 minutes, both distributed uniformly. The simulation experiment consists of 3 independent replications, and each replication will last for one working day, i.e. 8 hours.

The barbershop model in GPSS is given in Fig. 1. Line numbers at the right side are not the part of the model, and they are added for an easier orientation.

Transaction in the model represents a customer. Barbers are represented by a storage with capacity

defined explicitly (2) in the declaration part which starts with the keyword **simulate** (1).

```

simulate                                     1
barber storage 2                             2
                                             3
                                             4
                                             5
generate 12,6,1,400                          5
transfer 0.25,hairstyle                       6
shave queue qbarber                           7
enter barber                                  8
depart qbarber                                9
seize razor                                  10
advance 10,2                                  11
release razor                                 12
leave barber                                  13
transfer ,exit                                14
                                             15
hairstyle queue qbarber                       16
enter barber                                  17
depart qbarber                                18
advance 18,6                                  19
leave barber                                  20
exit terminate                                 21
                                             22
timer generate 60                             23
terminate 1                                   24
                                             25
                                             26
start 8                                        27
clear                                         28
start 8                                        29
clear                                         30
start 8                                        31

```

Fig. 1 Barbershop model in GPSS

Transactions are created and inserted into the model by the **generate** block (5) whose parameters "12,6,1,400" define: mean and spread of IAT, first arrival time, and maximum number of transactions generated, respectively. The **transfer** block (6) sends 25% of transactions (i.e. customers) to the haircut, and the remaining 75% proceeds to the next block for shaving.

The **enter** block (8) has to be used for the transaction to capture a place in the storage. In this case, capturing of a barber by a customer represents a beginning of servicing activity (i.e. shaving). **Queue** and **depart** blocks (7 and 9) are used to collect statistics about the waiting queues of customers. The queue will be automatically formed in the front of the barber storage when customers try to enter a busy barber.

The transaction uses **seize** block in order to capture a facility (10). The razor facility represents single razor that exists in the barbershop, and it is not (and cannot be) explicitly defined at the beginning of the model (in contrast with the barber storage (2)).

To simulate the duration of a certain activity (in the case of shaving), the **advance** block is used (11). Its parameters are the same as the first two parameters of **generate** blocks (mean and spread). After the shaving is completed, the customer will free resources razor (12) and barber (13) using appropriate blocks **release** (for facility) and **leave** (for storage), so another waiting customer could capture them afterwards. After that, the customer is unconditionally transferred (14)

to the exit of the barbershop, where he is destroyed and therefore removed from the model (21).

The blocks after the label haircut (16-21) are similarly organized, but for the haircut activity. Here, the razor is not used, and the service time is longer (19).

An independent simple segment is typically used for stopping the simulation (23-24). The transactions will be generated every 60 time units (i.e. every hour), and they will increase the termination counter by one (24) since the **terminate** block has 1 as its parameter.

The simulation is started with the control statement **start** (27), and its parameter 8 defines the value of the termination counter that will stop the simulation after 8 hours. **Clear** (28) is used to reset the collected statistics and to remove the transactions remaining in the model in order to prepare the model for the next replication. Two more replications are started by repeating the mentioned sequence (28-31).

Simple models, like the one presented here, are quite readable and easy to understand. Larger and more complex models quickly become less understandable and their structure is not so easy to follow.

3 Basic principles of GPSS++ design

The first question to answer in a language design is "Who will use the language and for what purpose?" Our primary goal was to develop a language that will be used in education, namely by students with the strong background in computing. The models that are to be written in GPSS++ will be small to medium sized. The second goal was to design a language that can be used in the wider community and will enable the development of large models.

The mentioned goals influenced all further decisions in language design. The list of the main decisions that we tried to follow during the development of GPSS++ is presented here:

- 1) GPSS++ should retain the original concept (one of the most important contributions of GPSS) of transactions and blocks as much as possible. As such it is appropriate not only for the education but also for the modelling of complex systems.
- 2) The syntax should be thoroughly modernised, and it should resemble the C-like syntax of contemporary languages. With this decision we consciously give up on backward compatibility, but for the educational purposes we do not see syntax changes as a great disadvantage. In addition, different versions of GPSS are also not compatible.
- 3) Simplicity of the language is very important because it reduces a learning curve for students or other potential users of GPSS++.
- 4) Modular and hierarchical modelling should be supported. It means that any part of a complex model can be separated as a standalone parameterised

submodel and used in other models. This request is crucial for the modelling of complex systems and achieving the better structure of the models of any size.

5) Although very flexible and powerful, the block structure of GPSS tends to produce very complicated and unstructured sequences of blocks (i.e. segments). In the previous section, the high-level model structure is addressed, and here the low-level structure of segments should be improved.

6) GPSS++ should allow hybrid modelling, i.e. mixture of continuous and discrete modelling. Even if it is not necessary for the educational purposes, some systems can naturally be expressed as hybrid models.

7) The flexibility of old GPSS is smaller compared to the modelling languages more similar to general-purpose programming languages (e.g. SIMSCRIPT). GPSS++ should extend the block structure with executable statements, like some versions of GPSS already did (e.g. GPSS World).

8) GPSS++ should have OO modelling abilities. The extent of the integration between "core-GPSS" and OOP is still under question.

9) GPSS is a special-purpose language and should be a high-level language, suitable for fast modelling and prototyping. The dynamic typing of GPSS is an advantage in that sense. Nevertheless, we think that GPSS++ should use static typing, because it improves readability and reliability of programs.

10) GPSS is very terse, since the keywords are used (almost) only at the beginning of a "statement". All arguments, conditions, etc. are given in form of a coma-separated list, which forces users to remember the meaning and position of parameters for each "statement". The syntax should be more verbose to help users in both reading and writing programs. On the other side, we think that English-like syntax can sometimes be more confusing than more formal and programming language-like syntax, so the compromise has to be made.

4 Non-modelling features of GPSS++

GPSS++ allows free formatting of a program text, unlike the old GPSS. It means that any block or statement can span over several lines of text, but also that a line of text can contain more than one block or statement. Comments can be single-line or multi-line and they follow the syntax of C++/Java.

The language is case sensitive. The number of leading characters in identifiers is neither prescribed nor restricted, and the same holds for the length of identifiers. The declaration part, segments, and experiment control parts are now clearly separated.

More details on some important language features are given in the following subsections.

4.1 Data types

There are a few basic built-in data types: integers (**int**), floating-point numbers (**float**), and Boolean values (**bool**). Two simulation-related built-in types are **time** and **state**. **Time** is similar to **float**, but its value is always positive and some operations act differently (e.g. the difference between two **time** values is always positive). **State** type is used in continuous modelling. Basically, **state** is **float** type with some internal data invisible to the programmer.

Some other types are not built-in, but rather library-based (e.g. String, File, List, etc.), although for a user this distinction is not important. User-defined types of GPSS++ are enumerations (**enum**), multidimensional arrays, classes, and generic classes (**template**).

4.2 Variables

Variables are similar to the variables in Java. Types with simple values, like **int**, **float**, **bool**, **enum**, **time**, and **state** have variables that contain the value itself. Arrays and objects have referential semantics, i.e. the variable holds the reference (pointer) to the real object. Objects are freed automatically if become unreachable (i.e. GPSS++ uses garbage collection). All variables are statically typed and they have well-defined initial values.

4.3 Expressions

Arithmetic expressions can use a standard set of operators: like +, -, *, /, %, ^, and mathematical functions like: trigonometric, logarithmic, square root, rounding etc. Logical expressions can use standard relational operators: ==, !=, <, <=, >, >=, as well as logical operators !, ||, &&, and ^. Logical operators have "short-circuit" semantics, as in C. It means that the second operand in the expression is not always evaluated (e.g. in the expression A && B, && is a logical-and operator; if the operand A is false, there is no need to evaluate the operand B).

4.4 Overall program structure

In order to prevent the namespace pollution, GPSS++ uses namespaces. Inside a namespace, a programmer can define all other entities of the language: functions, classes, models, experiments, variables, other namespaces, etc. All these entities can have public or private visibility. Qualified names are used to access public entities from other namespaces:

```
SomeNamespace.SubNamespace.publicName = 3;
```

Shorter writing is available by means of **using** directive that enables direct naming of entities:

```
using SomeNamespace.SubNamespace;
publicName = 3;
```

The program text can be divided into several files. Namespaces can span across several files, but the definitions of other entities cannot (e.g. functions, models, classes etc.).

4.5 Functions

A function in GPSS is actually a list of value pairs (x, f(x)). The GPSS functions defined in this way are very limited.

A variable in GPSS is similar to a macrostatement and it can be used as a very simple function, but it is restricted to a single arithmetic expression, without loop and branch statements. To pass parameters to a variable, a programmer must use savevalues (i.e. global variables), which could be very inconvenient. The term "variable" has completely different meaning in modern languages, and that is additional reason why GPSS++ has no equivalent feature. Variables in GPSS++ have the same meaning as in other programming languages, and they have nothing in common with variables of GPSS.

The GPSS-style functions are retained in GPSS++. They are called numerical functions. The numerical function has a slightly modified syntax, and its definition can be extended beyond the limits of the first or the last point (+INF in the following example). Fig. 2 shows continuous function f1 (dotted line) and discrete function f2 (single line), which can be defined like this:

```
continuous float f1 (float x) {
    1->2; 3->4; 4->3; 6->5; +INF
}
```

```
discrete float f2 (floatx) {
    1->1; 3->2; 5->2; 6->3
}
```

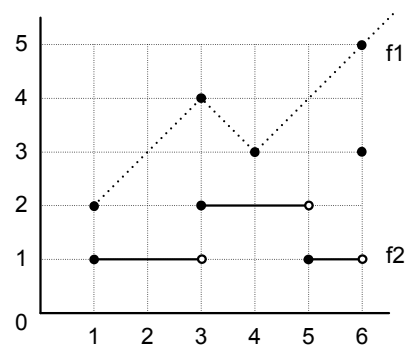


Fig. 2 Example of numerical functions

In addition, GPSS++ has functions like other programming languages. They are used to describe behaviour of a model or its parts. A function has zero or more call-by-value parameters. A function can return a value, or declare **void** as a return-type if it does not return a value. The definition of a mathematical function signum is shown here:

```
float signum ( float x ) {
    if ( x==0 ) {
        return 0;
    } else if ( x < 0 ) {
        return -1;
    } else {
        return 1;
    }
}
```

Functions are standalone if defined outside the class (e.g. inside model or namespace). Like in other OO languages, functions defined inside class are called methods and can be overloaded.

Since the functions are purely executable, they can have only statements; in contrast with the models which contain only GPSS-blocks (they are purely structural). GPSS++ also have a special kind of functions (named processes) used in continuous modelling.

GPSS++ does not have a main function, unlike C, C++, Java or Simscript. Starting points for the simulation are **generate** blocks inside the models, like in GPSS.

4.6 Classes

Classes in GPSS++ are alike classes in Java and C#. Class is a user-defined data type, and the instance of a class is called object. As usual, a class defines fields (i.e. member variables) and methods for its instances. Properties can be defined similar as in C#, and they are used as read and/or write accessors. Every part (i.e. field, method or property) of an object has **private**, **protected** or **public** visibility. Parts of a class declared by using keyword **static** belong to the class, and not to the objects.

A class can have a constructor, which is a special kind of function used for initialization of objects during their creation. Methods can be virtually overridden. Classes, methods, and properties can be **abstract** or **final**. GPSS++ allows definition of generic classes, similar to templates in C++. Many library-based classes are actually generic classes (e.g. List).

The following example shows a definition of `Customer` class that inherits `Transaction` class:

```
class Customer : Transaction {
    // fields
    public Time serviceTime;
    private int noOfServices [] = new int[5];

    // constructor
    Customer () { // constructor
        serviceTime = uniform(25,45);
    }

    // method
    public Count ( int service_type ) {
        noOfServices[service_type] ++;
    }
}
```

Objects (i.e. instances) of `Customer` class have two fields: publicly visible `serviceTime` used to store a service time for that instance, and private array of 5 integers named `noOfServices` where number of occurrence of each of 5 types of services will be recorded. A constructor initializes `serviceTime` variable by using random uniform distribution. The constructor is called automatically during the creation of an instance which takes place in the **generate** block (the next section explain this in more details). Public

method `Count` will be called upon completion of service, and the parameter indicates one of five possible service types.

Single inheritance is supported in the current version of GPSS++. Here are few examples which illustrate the need and the application of multiple inheritance in the modelling of transactions.

In the imaginary model of a university transactions can represent real persons. Persons can be divided in students and employees. The further division of students can be made to graduate and postgraduate students. Employees can be divided to teachers, technical staff etc. In order to model a teaching assistant who is postgraduate student and employee at the same time, multiple inheritance is preferred and greatly simplifies the modelling process.

Next example will illustrate the need and the appropriate application of multiple inheritance in traffic. Transaction types `Car`, `Truck`, `Van`, and `PriorityVehicle` are defined as subclasses of `Vehicle` class. So, the `PoliceCar` class can be modelled as a subclass of `Car` and `PriorityVehicle` classes, and the `FiremanTruck` class can be a subclass of `PriorityVehicle` and `Truck` classes.

Similar reasons can be found in the modelling of bank transactions, which are divided into `Deposit`, `Withdrawal`, etc. The `MoneyTransfer` class, that represent transfer from one account to another, can be subclass of both classes - `Deposit` and `Withdrawal`.

We should mention that the multiple inheritance, together with other extensions to GPSS++, could enable a mixture of transactions and simulation entities, i.e. dynamic and static entities of simulation. For example, a `Truck` class can be viewed as a facility in the transport simulation, but it can be viewed as a transaction when it comes to the servicing of the mentioned `Truck`.

5 Transactions and blocks

Transactions in GPSS can have a number of parameters that need not be declared in advance. They are used as needed, and the simulator will dynamically create them. As we already mentioned, this can be a desirable feature for a high-level language, but it can also easily introduce hard-to-find errors, and reduce the readability of a program.

Transactions in GPSS++ are just a kind of objects and they are defined by using classes (they have to inherit the `Transaction` class). The `Transaction` class has most properties of transactions in GPSS. For example, it defines fields `Priority`, `CreationTime`, `Lifetime`, etc. GPSS++ uses object's fields instead of the GPSS parameters. In addition, GPSS++ offers methods, properties and constructors for the transactions.

Since the transactions have to be explicitly defined, a GPSS++ model is typically bigger than its counterpart

in GPSS. But, the GPSS++ model is much more understandable, readable, and better self-documented. Later, we shall see that transactions are not the only part of the language that should be explicitly defined in GPSS++.

Most of the blocks from GPSS can be found in GPSS++. Some of them are modified for the same reasons as in WebGPSS, namely for the simplicity and ease of learning. Other blocks are enhanced with some abilities for which we believe that they will be useful in modelling. In general, we dismissed the syntax where parameters of blocks were given as a comma-separated list, which we found hard to read. Several GPSS++ features are explained in the following subsections in order to illustrate the changes.

5.1 Generate block

Like in old GPSS, **generate** blocks create transactions. An example shows the block that generates `Vehicle` transactions with inter-arrival time uniformly distributed between 2 and 10 time units. The number of generated vehicles is restricted to 12000, and the first vehicle is generated after 200 time units. Presumably, `Vehicle` is a class of transactions defined elsewhere:

```
generate upto 12000 Vehicle
      after 200 every 6 +- 4;
```

We think that such syntax is much more obvious and natural than in old GPSS (`generate 6,4,200,12000`), and especially in the case of omitting some parameters. (`generate 6,,,12000`).

Additionally, in GPSS++ we can easily define the generation of several types of transactions, e.g. we can define that among `Vehicle` transactions there are 80% of cars, 13% of busses, and 7% of trucks (here, besides `Vehicle` class, we had to define three classes: `Car`, `Bus`, and `Truck` that are inherited from `Vehicle`):

```
generate upto 12000 Vehicle
      ( Car: 80; Bus: 13; Truck:7 )
      after 200 every 6 +- 4;
```

5.2 Facilities and queues

GPSS requires no explicit definition for the most simulation entities, like facilities, queues, savevalues (i.e. global variables), etc. When such entity is used for the first time somewhere in the model, it will be created. In GPSS++, everything has to be defined in advance, and that also holds for the simulation entities which are nothing more than objects, i.e. the instances of the predefined classes.

In the next example, the facility named `Barber` is defined. Afterwards, we can use blocks **seize** and **release** like in old GPSS. To gather statistics about the waiting in the queue in the front of `Barber`, we can additionally define a queue named `barberQueue`. Blocks **queue** and **depart** are called **inqueue** and **outqueue** in GPSS++, but they can be replaced by a **forming**-expression (similar to WebGPSS):

```
Facility Barber = new Facility(priorityFifo);
Queue BarberQueue = new Queue();
...
generate Customer every exponential(880);

seize Barber forming BarberQueue;
advance 600+-120;
release Barber;

terminate;
```

Transactions trying to capture certain facility form an implicit queue. The default queue order is FIFO with priority (like in this example), but it can also be FIFO without priority, LIFO with or without priority, or FIRO (first-in random-out) with or without priority.

GPSS uses SNAs (system or standard numerical attributes) as a mean of accessing the different values. For this example, let us just mention SNA `F` used to check whether a facility is free or busy. Instead of using cryptic syntax `F$Barber`, GPSS++ uses simple OO-style: `Barber.Free` or `Barber.InUse`.

Furthermore, GPSS++ allows the dynamic change of queue order and the dynamic change of storage capacity.

5.3 If block

The structure of blocks in GPSS is hard to follow since it uses numerous jumps, similar to "spaghetti-code". In order to obtain a better structure of models we have introduced an **if** block:

```
if( Vehicle.route == Direction.left ) {
  seize left_road;
  advance 12 +- 2;
  release left_road;
} else {
  seize right_road;
  advance 20 +- 5;
  release right_road;
}
```

In the old GPSS this example would be implemented by using **test** block, **transfer** block and labels. Note that the branches of the **if** block can contain only another blocks. However, we did not prohibit transfer of the transactions to any label inside the model. For that purpose, a **goto** block is used.

5.4 Wait block

Wait block is introduced to replace the GPSS blocks that operate in the refusal mode (e.g. **gate**, **test**, etc.). It has two forms: **wait while** and **wait until**. The waiting condition has some restrictions, but this is beyond the scope of this paper. Here is an example of the **wait** block that will hold transactions until a place is freed in the `Parking` storage but with a limited waiting time of 120 time units. Additionally, queue statistics is gathered by using `ParkingQueue`:

```
wait until (Parking.NotFull)
      forming ParkingQueue
      timeout 120;
```

Wait allows a more complicated behaviour definition. For example, one can write:

```

wait until (condition) then {
    ... // blocks "executed" if a transaction
        // has passed after some waiting
} else {
    ... // blocks "executed" if a transaction
        // has passed without waiting
} timeout 120 {
    ... // blocks "executed" if a transaction
        // has passed due to a timeout
}

```

5.5 Execute block

The old GPSS uses blocks **savevalue** and **assign** to store a value in the savevalues and transaction parameters. Another usual task is to compute something during the transaction movement. This is either complicated in GPSS, or requires the external procedures written in FORTRAN. GPSS++ uses the **execute** block for all these purposes. After **execute** block, there can be any sequence of statements written inside braces. The **execute** block is activated when a transactions enters it. For example:

```

// Vehicle enters the road section
seize Road_section_332;

execute {
    // instead of savevalue
    vehicles_count = vehicles_count + 1;

    for (...) {...} // any statements

    // instead of assign
    Vehicle.km = Vehicle.km + 2.3;
}

advance 140 +- 35;
release Road_section_332;

```

6 Submodels

A user can define a named sequence of blocks that can be used in other parts of a model as a submodel. A submodel can have its own local variables, functions, storages etc., it can have more than one entry end exit point for the transactions, and it can use parameters. An example of a very simple submodel and its usage is shown next:

```

// "submodel"
public model machine_operation
( Facility machine, Time operation_time ) {

    seize machine;
    advance operation_time;
    release machine;
}

// "supermodel" - factory with 2 machines
public model factory () {
    define {
        Facility machine[]=new Facility[2](Firo);
    }

    generate Products every 40 +- 10;
    machine_operation( machine[0],
        exponential(38) );
    machine_operation( machine[1],
        exponential(16) );
    terminate;
}

```

7 Continuous modelling

Continuous part of a model is defined by a set of differential equations placed inside the special kind of function, called process. The process is started and stopped explicitly using the statements **start** and **stop** (it can be also started automatically). Processes can be defined inside objects, classes, models, or namespaces. The given example depicts a system where the transactions represent ingots arriving to the oven for heating:

```

class ingot : Transaction {
    // temperature of the ingot
    public state temp;
    // size of the ingot
    public int size;

    // continuous process of heating
    public process void HeatUp
        (float ovenTemperature) {

        derivation
            temp = 0.027 * (ovenTemperature - temp);
    }

    ...
    // somewhere in the model: ingots arrive
    // entering the Oven storage
    enter ingot.size units in Oven;
    execute { start ingot.HeatUp(800); }
    wait until ingot.temp > 600;
    execute { stop ingot.HeatUp; }
    leave ingot.size units in Oven;
}

```

The variables used in differential equations have to be defined as **state**. Processes can communicate through non-local **state** variables, thereby forming a structure of connected continuous subsystems.

8 Statistics

The old GPSS uses queues with the **queue/depart** blocks and tables with the **mark/tabulate** blocks in order to gather the statistics. Besides the renaming the **queue/depart**, we removed the **mark/tabulate** blocks and added traced variables, conceptually more similar to the **accumulate** and **tally** statements of Simscript [12]. Traced variables are sampled on every change, and the samples can be weighted by time (**trace continuous**) or non-weighted (**trace discrete**). The following statistics can be obtained: mean, variance, standard deviation, maximum, minimum, histogram, etc.

Random numbers can be generated by using a default generator, or by creating the arbitrary number of independent generators. Such generator is simply an object of an `RND` class. GPSS++ also have built-in exponential, normal, uniform, triangular, Poisson, and discrete uniform distributions.

9 Simulation experiments

GPSS++ uses specific statements placed in a simulation block for the control of the simulation experiment. Every model that should be simulated is

defined by **configuration** statement that loads the model. Afterwards, the simulation is started with the **run** statement:

```
simulation {
  // load model parameterised by 22
  configuration BusStation(22);

  // warm up for 1 hr (3600 seconds)
  run upto 3600;

  // reset statistics, leave transactions
  reset;

  // simulate for next 100 hrs
  run upto 360000;

  // load model parameterised by 32
  configuration BusStation(32);

  // simulate 5 times
  run 5 simulations upto 3600;
}
```

10 Example of a barbershop in GPSS++

The model from the section 2 is presented here, but this time written in GPSS++ (Fig. 3). It could be written more similarly to the original, but we wanted to illustrate some new features of GPSS and make the model well structured and more object-oriented.

The model in GPSS++ is not restricted to the sequence of the declaration part, segments, and the experiment control. Any part can be placed in a separate file, or the parts can be combined in a single file. For a simple model as the barbershop model, we put all parts in a single file.

At the beginning, the `Customer` class is defined (1-11) as **abstract** since its instances are not going to be used in the model. `Customer` is a subclass of the `Transaction` class, and hence its (indirect) instances will be used as transactions in the model. The `Customer` class defines two private fields for storing the mean and the spread of the service time (3). The property `serviceTime` (5-10) is used to obtain a random value and it uses fields `mean` and `spread` to define the parameters of uniform random distribution.

Two concrete classes, `ShaveCustomer` (14-19) and `HaircutCustomer` (22-27), inherit the `Customer` class. They represent two types of customers. They extend their superclass with the constructors (16-18 and 24-26). The constructors simply initialize the `mean` and `spread` fields to the appropriate values (17 and 25).

The `barbershop` model is defined next (31-58). It is parameterized with a number of barbers, and with the duration of simulation (31-32). The definition block (33-37) is at the beginning of the model. It contains the definitions of all static simulation objects. Here we can see the explicit definitions of the facility and the queue, which were not present in the GPSS model.

The remaining of the model consists of segments (39-57). The **generate** block is much longer, and much

more readable than in old GPSS. The central part of the model is not implemented using **transfer** blocks that produce spaghetti-code. Instead, the better structure on block-level is achieved by using **if** block, similar to well-known if-statement. Two branches of **if** block represent two different types of services. The branches itself are quite similar to the original model, and the same holds for all the blocks in the model.

```
abstract class Customer : Transaction {      1
  2
  private Time mean, spread;                3
  4
  public Time serviceTime {                 5
    read {                                   6
      return uniform( mean-spread,          7
                     mean+spread );        8
    }                                       9
  }                                       10
}                                       11
 12
class ShaveCustomer : Customer {           13
  14
  ShaveCustomer () {                         15
    mean = 10; spread = 2;                  16
  }                                         17
}                                         18
 19
class HaircutCustomer : Customer {         20
  21
  ShaveCustomer () {                         22
    mean = 18; spread = 6;                  23
  }                                         24
}                                         25
 26
model barbershop ( int NoOfBarbers,       27
                   int Timeout ) {        28
  29
  define {                                    30
    Storage barber = new Storage(2);        31
    Queue qbarber = new Queue();           32
    Facility razor = new Facility();        33
  }                                         34
  35
  generate upto 400 Customer                36
    ( HaircutCustomer : 0.25,              37
      ShaveCustomer : 0.75 )              38
    after 1 every 12+-6;                   39
  40
  enter barber forming qbarber;             41
  42
  if(Customer instanceof ShaveCustomer){   43
    seize razor;                            44
    advance Customer.serviceTime;          45
    release razor;                          46
  } else {                                  47
    advance Customer.serviceTime;          48
  }                                         49
  50
  leave barber;                             51
  terminate;                                52
  53
  generate after Timeout;                   54
  terminate;                                55
}                                         56
  57
simulate {                                  58
  configuration barbershop(2,480);         59
  run 3 simulations;                       60
}                                         61
  62
  63
  64
  65
```

Fig. 3 Barbershop model in GPSS++

The stopping segment (56-57) uses the `Timeout` parameter of the model to define the end of the simulation.

Finally, the simulation experiment is defined in a separate block (62-65). The first statement (63) defines the model which will be simulated. Parameters define that 2 barbers will be used, and the simulation duration will be 480 time units. The second statement (64) begins the simulation that will consist of three independent replications.

A short comparison shows that the original model in GPSS has 25 lines of code, while the GPSS++ model has effectively 31 lines (without lines with closing brace only, and without lines that are wrapped due to a line length limit).

11 Comparison of GPSS and GPSS++ models

For a comparison we took about 50 small-sized examples from the widely known and available sources [19, 3]. Then we implemented the same models in GPSS++. We found that in most cases GPSS++ programs are longer than the originals (about 20%), especially for “small” models (about 50 lines), where explicit definitions required by GPSS++ add significant amounts of code. If the original model uses some of the features omitted from GPSS++, the implementation of such features can increase the program size and its complexity by 80%. In the case of “bigger” models (about 150 lines), the advantages of GPSS++ became more obvious. Submodels reduced the code size by 20%, and sometimes even by 50%, depending on the system implemented.

Obviously, the main advantage of GPSS++ is not the code reduction. Instead, the better organization, improved readability and modularity, easier modelling, debugging etc. are the major improvements of GPSS++ over previous versions of the language.

12 Conclusion and future work

GPSS has been successfully used for many years in the modelling and simulation of discrete-event systems. We have proposed several improvements of GPSS that should make it more appropriate for today’s programmers, but at the same time we have tried to keep the concept of transactions and block structure.

The main improvements are: modernized and more verbose syntax, static typing, modularity and hierarchical modelling, features of general-purpose language for modelling complex behaviour and algorithms, hybrid modelling of discrete and continuous systems, and finally object-oriented modelling.

Currently, the GPSS++ tools are still under development, and for now we plan to implement only

compiler and simulator without a support for visual modelling, animation, graphical presentation of results, etc. The language itself is also under development. We can mention a few problems noted so far and possible future improvements.

Firstly, some GPSS blocks are omitted (e.g. **link**, **unlink**, **select**, **scan**, **match**, **buffer**, **preempt**) together with some options available with particular blocks (e.g. **both**, **pick**, and **all** options for **transfer** block). Here we did not count for omitted features that are replaced or can be easily implemented using the new features of GPSS++. The missing functionality can be implemented in most of the cases by using the **execute** block, but the implementation is not always simple enough. Therefore, some of the omitted blocks (e.g. **preempt**) should be included in the next version of GPSS++.

GPSS++ has no support for the user-defined blocks, although the similar effect can be achieved using submodels (but not to the full extent). Namely, a submodel cannot use other block segments enclosed in the braces, like built-in blocks (e.g. branches of an if block or wait block). Also, it is not possible to use call-by-reference or call-by-name parameters with functions and submodels. The mentioned features would significantly improve the flexibility of modelling.

Another missing feature is related to the constructors of transactions. Namely, constructors of transactions cannot have parameters. Now, in the case that parameters are required, additional call has to be made after the transaction leaves a generate block. It should be possible to call a constructor with parameters immediately in the generate block.

Next, we plan to add a multiple inheritance. It could be useful in general cases if a class have the properties of two or more existing classes, but also for the modelling, as explained in section 4.6.

Using functions as first-class values, i.e. as parameters and as results of other functions, is supported in functional programming paradigm, and possible in languages with pointers to functions (e.g. C). Such usage of functions would be of great help in some situations and this is currently not supported in GPSS++. The old versions of GPSS are able to return a function as a result of other function (it can even return a label as a result). Of course, it is not a consequence of GPSS being supported functional programming paradigm, but rather its closeness to the assembly language level in some aspects.

Finally, the level of the integration between old GPSS and new OO and general programming language features is still under question. In the current version of GPSS++, the border between the two is quite clear. We will investigate the possibility of closer integration with no giving up on the main concepts of GPSS.

13 References

- [1] G. Gordon. System Simulation, Prentice-Hall. 1969.
- [2] T. J. Schriber, S. Cox, J. O. Henriksen, P. Lorenz, J. Reitman, and I. Ståhl. GPSS Turns 40: Selected Perspectives. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceedings of the 2001 Winter Simulation Conference*, 2001 Winter Simulation Conference, pages 565-576, December 9-12 2001. Arlington, USA.
- [3] http://www.minutemansoftware.com/tutorial/tutorial_manual.htm
- [4] <http://www.webgps.com>
- [5] I. Ståhl. From 44 to 31 to 28 to 22 and Now to 18 Less Becomes More in GPSS. In T. Schulze, S. Schlechtweg, V. Hinz, editors, *Simulation und Visualisierung 2003 (SimVis 2003)*. Simulation and Visualisation Conference, pages 465-478, March 6-7 2003. Otto-von-Guericke-Universitat, Magdeburg, Germany.
- [6] I. Ståhl. GPSS - 40 Years of Development. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, *Proceedings of the 2001 Winter Simulation Conference*, 2001 Winter Simulation Conference, pages 577-585, December 9-12 2001. Arlington, USA.
- [7] J. O. Henriksen. An Introduction to SLX. In S. Andradottir, K. J. Healy, D. H. Withers, and B. L. Nelson, editors, *Proceedings of the 1997 Winter Simulation Conference*, 1997 Winter Simulation Conference, pages 559-566, December 7-10 1998. Atlanta, USA.
- [8] O.-J. Dahl and K. Nygaard. Simula - an ALGOL-Based Simulation Language. *Communications of the ACM*, 9:671-678, 1966.
- [9] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Some Features of the Simula 67 Language. In J. Reitman, J. Waxweiler, H. Falk, and A. Ockene, editors, *Proceedings of the second conference on Applications of simulations*, Conference on Applications of Simulations 1968, pages 29-31, December 2-4 1968. New York, USA.
- [10] B. Stroustrup. The C++ Programming Language. Addison-Wesley. 1991.
- [11] J. A. Joines and S. D. Roberts. Fundamentals of Object-Oriented Simulation. In D. J. Medeiros, E. F. Watson, J. S. Carson, and M. S. Manivannan, editors, *Proceedings of the 1998 Winter Simulation Conference*, 1998 Winter Simulation Conference, pages 141-149, December 13-16 1998. Washington, USA.
- [12] S. V. Rice, H. M. Markowitz, A. Marjanski, and S. M. Bailey. The SIMSCRIPT III Programming Language for Modular Object-Oriented Simulation. In M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, editors, *Proceedings of the 2005 Winter Simulation Conference*, 2005 Winter Simulation Conference, pages 621-630, December 4-7 2005. Orlando, USA.
- [13] D. R. Kalasky and G. A. L'vasseur. Using Simple++ for Improved Modeling Efficiencies and Extending Model Life Cycles. In S. Andradottir, K. J. Healy, D. H. Withers, and B. L. Nelson, editors, *Proceedings of the 1997 Winter Simulation Conference*, 1997 Winter Simulation Conference, pages 611-618, December 7-10 1998. Atlanta, USA.
- [14] P. L'Ecuyer and E. Buist. Simulation in Java with SSJ. In M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, editors, *Proceedings of the 2005 Winter Simulation Conference*, 2005 Winter Simulation Conference, pages 611-620, December 4-7 2005. Orlando, USA.
- [15] B. W. Kernighan and D. M. Ritchie. The C Programming Language. Prentice-Hall. 1988.
- [16] K. Arnold and J. Gosling. The Java Programming Language. Addison-Wesley. 1996.
- [17] [http://msdn2.microsoft.com/en-us/library/618ayhy6\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/618ayhy6(VS.80).aspx)
- [18] T. Pawletta, S. Drewelow, S. Pawletta. Discrete Event Simulation in Interactive Scientific and Technical Computing Environments. In R. N. Zobel and D. P. F. Möller, editors, *Proceedings of the 12th European Simulation Multiconference*, 12th European Simulation Multiconference, pages 529-533, June 16-19 1998. Manchester, UK.
- [19] T. J. Schriber. Simulation Using GPSS. John Wiley & Sons. 1974.