HGPSS: A HIERARCHICAL EXTENSION TO GPSS

Filip Claeys, Hans Vangheluwe, Ghislain Vansteenkiste
Department of Applied Mathematics, Biometrics and Process Control
University of Gent
Coupure Links 653, B-9000 Gent, Belgium
Hans.Vangheluwe@rug.ac.be

INTRODUCTION

In recent years, a plethora of new computer and simulation languages have emerged. Though one can argue whether such diversity is really needed, the introduction of structured and object-oriented languages is definitely an improvement. On the other hand, one important reason for the continued use of existing modelling and simulation languages is their inherent quality and widespread use. GPSS [Sch74, Gor78] is a perfect example of a language attractive because of its expressive power and the existence of a graphical representation. Recently, a need was felt for a full-scale public domain process-interaction discrete event simulation system. Such a system would allow experimentation at source code level and complex simulations would become possible without major investments.

This article describes the development of a hierarchical extension to the well-known process-interaction discrete event modelling and simulation language GPSS. The requirements for an ideal discrete event modelling language are analysed. The reasons for choosing GPSS as a starting point are explained. After evaluation of the differences between the ideal situation and GPSS, a hierarchical extension to GPSS named HGPSS [Cla92a, Cla92b], is proposed. The language design is described and some examples of the key features are given. This covers both additions to GPSS and modifications to the language. The software architecture which consists of a HGPSS language processor and a C++ simulation kernel is discussed.

GENERIC MODELLING AND SIMULATION LANGUAGE REQUIREMENTS

In the design of a modelling language, some generic requirements can be identified. The requirements listed below were at the basis of the design of HGPSS. Future modifications of HGPSS will also be driven by these “theoretical” considerations.

A modelling language should have constructs for expressing the generic aspects of the model (as to allow for generic manipulation of model objects) as well as for expressing the type specific aspects of the model (which will allow for explicit simulation of the model). Extension towards new abstractions should not require a modification of the generic parts of the language.

The language should be both syntactically and semantically formally defined. Furthermore, the language must contain constructs to elicit meaning (semantics) of models. The meaning of models can be assessed through

- a specification of the syntax – semantics mapping (e.g., ontologies)
- reference to context

Actually, making semantics explicit is necessary at two levels: on the one hand, the semantics of the modelling language as such and on the other hand, the semantics of the models. The modelling language should support constructs for making the semantics of models explicit (e.g., through units).

The language should allow for model structuring. It should at least support hierarchical sub-model (de)composition (to arbitrary depth, including information hiding – encapsulation–).

The language must be modular in order to support re-use of models and model components. Usually, modularity refers to functional decomposition as opposed to structural decomposition. The hierarchical (de)composition mentioned above may be used for either. Re-use of models refers to the portability of a module across different contexts. In porting models there are two main issues:

- The choice of the model to be re-used: the language must provide constructs to allow an external agent to choose the appropriate model without inspecting its contents in detail.
- The usefulness of the model in different contexts.
In general, compatibility of INTERFACES may be sufficient. In practice, causality (in the case of continuous models) and type differences (continuous vs. discrete) of the state space may pose difficulties. A strategy for resolving these difficulties must be provided.

Related to semantics is the inclusion of redundant information for documentation and consistency checking. In manipulating models, consistency should be proven (or at least checked) wherever possible. This is the price one has to pay if models are to be meaningfully re-used.

The modelling language must be useful for model documentation, describing both its structure and behavior, thus serving as a standardised exchange language between modellers and modelling tools.

The modelling language must be suitable for both user and tool use. In case of user use, a graphical representation (especially with respect to hierarchical decomposition) may enhance usability (BUT the SAME information MUST be present in the non-graphical –textual– representation). In case of tool use, distinction can be made between two types of tools:

- **Modelling tools** allow for the manipulation of models. This requires an internal representation of the model (the modelling language is just one external representation) together with persistent versions in a Model Base. The internal representation will be close to the formal specification. All modelling tools should at least support reading in an external (textual) representation, and subsequently writing it out without information loss (may not be syntactically identical however). This ensures:
  - Model portability (across widely varying platforms/tools). In particular, in a client/server architecture, the minimally available communication medium will be ASCII text transfer (e.g., e-mail, http).
  - The possibility of drastically changing modelling tools and their implementation
  - Human readability of models which may have been constructed automatically or by means of a sophisticated graphical tool.

- **A simulation tool** (either symbolic or numerical) takes a model and generates system behavior given certain experimental conditions. As opposed to modelling tools, simulation tools are generic modelling language independent BUT model TYPE dependent.

The distinction between modelling and simulation tools is vague with respect to representation: both a modelling and a simulation language represent a “model” of a system. With respect to operations, modelling languages allow symbolic manipulation of the model and try to capture as much information as possible. Simulation languages are focused on “running” simulations where context information is less important and execution speed is of prime importance.

The following are some essential parts of the modelling and simulation process:

- Hierarchical model development (both top-down, bottom-up and combinations) must be supported to manage complexity of the “lots of components” type.
- Multi-abstraction modelling is necessary to manage complexity of the “different abstractions” type. This involves coupling models in different abstractions, transformation between different abstractions and iso(homo)morphism between models.
- Incremental model development must be possible:
  - by refining existing models (hierarchically)
  - by adding to models
  - by modifying models
- The concretisation of models: i.e., making system behavior explicit through simulation
- Storing and retrieving models (in its simplest form: ASCII files)
- Distributed (no centralised data model) model sources support, possibly based on a client/server implementation model. The lack of a centralised data model (apart from the generic structure of the modelling language) is an important issue which needs to be resolved with respect to semantics.
- A transformational approach must be supported: specification → implementation, which in turn is interpreted as a specification. . . Such an approach closely follows theoretical models of software development.
- Whenever a choice is possible, favour model re-use over efficiency.

**HGPSS LANGUAGE DESIGN**

GPSS was originally developed by Gordon at IBM in 1962. The language has been used extensively and different commercial implementations were developed ever since its conception. Despite its expressive power and flexibility, the language suffers from some weaknesses. The major deficiencies are the lack of conceptual support for hierarchical modelling, the poor interfacing facilities with other software
components and the absence of procedures altering the way the GPSS processor works.

In HGPSS, some constructs were added or modified with respect to the original GPSS. All adaptations were inspired by the need to satisfy the generic modelling language requirements described above and to eliminate the GPSS flaws. The HGPSS-specific features can be summarised as follows:

- Modularity has been enforced by explicitly separating the system model and the experiment [Zei76]. The system model is described in the so-called model section while experiment code is listed in the command section (Figure 1).

```
(1) MODEL TheSystem(ParameterType parameter)
(2) ...
(3) ENDMODEL
(4)
(5) COMMAND
(6) ...
(7) SIMULATE TheSystem(n)
(8) ...
(9) ENDCOMMAND
```

Figure 1: Model and command section

- A model section can be made up of one or more, possibly parameterised sub-model sections. Sub-model instances can be declared within other sub-models, thus allowing for a sub-model hierarchy to be built. Figure 2 shows a model (Model1) containing two sub-models (Model2 and Model3). A transaction is first routed to Model2 and subsequently to Model3.

- The language can be extended using user-defined blocks. Those are blocks without any initial functionality which can be tailored to certain needs by supplying C++ code for pre-defined event procedures.

- Models can interact with external software components through user-defined interfaces implemented by C++ functions. Those procedures can be set to run at each event time or just in case a transaction flows through a specific block.

- Some further minor changes have been made to the original GPSS. Those allow for more self-explaining labels and comments to be used. Furthermore, it is possible to embed C++ code in HGPSS programs and to refer to C++ variables and functions from within HGPSS statements. Some additional facilities for specifying original GPSS functions and variables have also been implemented.

```
(1) MODEL Model1
(2) A SUBMODEL Model2
(3) B SUBMODEL Model3
(4) ...
(5) ENTERMODEL A,In
(6) LEAVEMODEL A,Out
(7) ENTERMODEL B,In
(8) LEAVEMODEL B,Out
(9) ...
(10) ENDMODEL
(11) MODEL Model2
(12) INPUT In
(13) ...
(14) OUTPUT Out
(15) ENDMODEL
(16) MODEL Model3
(17) INPUT In
(18) ...
(19) OUTPUT Out
(20) ENDMODEL
(21) COMMAND
(22) ...
(23) SIMULATE Model1
(24) ...
(25) ENDCOMMAND
```

Figure 2: A multi-model system

THE ARCHITECTURE

The HGPSS system as a whole can be regarded as a two-level system.

- The bottom level is hosted by a C++ object-oriented programming environment. Within this environment a discrete event simulation kernel has been built which consists of a number of interrelated functions and classes supporting GPSS-like process-interaction simulation. Calls to this kernel are implemented by a multitude of standardised interface functions. For obvious reasons, this interface language has been named HGPSS++.

- The top level is made up of a precompiler converting HGPSS program statements into the set of corresponding HGPSS++ kernel calls.

Figure 3 depicts the process of generating a simulation executable using HGPSS. It shows the system to be simulated can be modelled using any mixture of partial HGPSS (high-level), HGPSS++ (mid-level) or C++ (low-level) descriptions. The HGPSS precompiler converts HGPSS statements into HGPSS++ function calls which are subsequently compiled into object code. The latter also applies to any pure C++ code. Compiled sub-models can be filed in a sub-model library or linked with the HGPSS++ kernel in order
to construct a simulation executable.

**HGPSS++ KERNEL DESIGN**

The requirements for a simulation kernel are entirely different from those of a modelling language. Whereas the modelling language needs to be expressive and meaningful, a simulation kernel must be modular and performant. In the current HGPSS++ simulation kernel implementation however, very little attention was paid to the performance issue. The modularity of the kernel was considered of higher priority. The rationale for this is the fact that by nature, GPSS is very well-suited for the simulation of systems of high structural complexity involving relatively few events. This means for most systems performance is of only minor concern. Event scheduling-based networking simulation systems on the other hand, do heavily rely on performance.

In the ideal situation, simulation practitioners would like simulation kernels to be sufficiently modular to allow for plug and play. Though modelling environments may be vastly different in nature, the executable simulation should be easy to interface with. In particular, different simulators should be easily interfaceable. This is currently not the case with many commercial simulation systems. Often, the only standard way of exchanging state information is via files!

A recent standardisation effort in the continuous simulation domain introduces the concept of a DSblock (Dynamic System block) [Ott92]. It identifies the basic components of a system model when it is being simulated, i.e.:

- The inputs and outputs of the system
- The internal state of the system
- The parameters and initial conditions
- The kernel functions used to “drive” the simulation experiment (i.e., to calculate the evolution of the state vector through time).

The DSblock approach proposes a standardised interface to a Dynamic System block. All simulation systems should adhere to a “standard” interface definition resulting in an *Application Programmer’s Interface* (API).

In HGPSS++ an attempt was made to provide the basic DS-block functionality for discrete event systems. This is realised through a well-defined C++ interface providing access to the above mentioned basic system components.

The HGPSS++ kernel has been designed as a hierarchical set of classes organised around a supervising Processor class. Most classes model the original GPSS entities and blocks, or those additionally defined in HGPSS. The core of the processing is based on the scheduling of events on a so-called future and current event chain.

**APPLICATIONS**

HGPSS can be used to solve all traditional problems for which GPSS has proven useful before. Its hierarchical modelling capabilities furthermore allow to tackle problems of high structural complexity which would be very difficult to handle using the original GPSS. Because of its open architecture and sub-model library capabilities, HGPSS can be turned into a dedicated simulation environment.

As the “process” of software development may be concurrent (non-sequential), time dependent and even stochastic (when the user is modelled), representing (and even simulating) that process [HK89] should be done using an abstraction capable of expressing these properties of the “software process”. A process interaction discrete event abstraction seems most appropriate. This was actually the original reason for the development of HGPSS: to have a modelling language/simulation system capable of expressing complex software processes. Currently, HGPSS is being used extensively for software project planning and management.

**HGPSS AND OTHER SIMULATION LANGUAGES**

One might be wondering how HGPSS compares to other simulation languages. In the following, HGPSS will therefore be briefly compared to SIMSCRIPT and MODSIM. Both of these are process-interaction discrete event simulation languages, just like HGPSS. There is no point in comparing to any other kind of language, since this will not allow for sensible conclusions to be drawn.

SIMSCRIPT uses its own English-like scripting language and is not based on any general purpose host language. SIMSCRIPT programs are generally longer than GPSS programs and more difficult to understand. Graphical representation is not as straightforward as in GPSS. Because of the rigid division between an experiment section and one
or more process model sections, SIMSCRIPT is well-suited to tackle problems of considerable size. Since the problems the original GPSS had in handling complex systems are largely overcome by HGPSS, this is not an advantage of SIMSCRIPT over HGPSS.

MODSIM is based on MODULA-II but is not a perfect superset of this language. As in SIMSCRIPT, there is no straightforward graphical representation and programs tend to be lengthy as well as difficult to understand.

FUTURE DIRECTIONS

It is intended to build an integrated development environment around the HGPSS system consisting of a graphical model building tool, debugger and editor. In addition, HGPSS will be used to represent discrete event systems in a multi-paradigm model specification language. By further implementing the DSBLOCK compatible interface to the HGPSS kernel, it should be possible to plug and play the kernel in various contexts. At last, some work will be done to optimise performance, although this is of no major concern. In the end, the HGPSS software will be made publicly available via electronic means.

References


