MODELLING AND SIMULATION 1994

ESM94

EDITED BY
Dr. Antoni Guasch
and
Dr. Rafael M.Huber

JUNE 1-3, 1994
UNIVERSITAT POLITECNICA DE CATALUNYA
BARCELONA, SPAIN

A Publication of the Society for Computer Simulation International

Proceedings of the Conference on Modelling and Simulation 1994
eds. Guasch and Huber

DESIGN OF A USER FRIENDLY MODELLING AND SIMULATION ENVIRONMENT

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

Vladimir Visipkov, Yuri Merkuryev, Galina Merkuryeva, Artis Teilans
Department of Modelling and Simulation,
Riga Technical University
Kalku Street 1, LV-1658, Riga, Latvia
vlad@itl.rtu.lv

ABSTRACT

The design of a modelling and simulation environment and aspects of its implementation are described. The GISMOS (Generic Interactive System for MOdelling and Simulation) design offers a conceptually clean modelling and simulation method, applicable to a wide range of model formalisms. Semantics of models is made explicit through the use of ontologies. All knowledge regarding a system is encapsulated in models, which are manipulated appropriately to satisfy goals. Both the model descriptions and the modelling/simulation process are formalised. The current GISMOS design focuses on modelling and simulation of discrete-event systems, and in particular on the optimisation of such systems. Tactical and strategical design of simulation experiments are presented as an instantiation of the generic modelling and simulation process.

INTRODUCTION

The analysis, design and control of complex systems (hardware, software and hybrid) involves the manipulation of different abstract representations or "models" of these systems. Typical abstractions used in physical systems modelling are bond graphs, petri-nets, differential equations and queuing networks. In software systems, abstractions include finite state automata and entity relationship diagrams. The representation and manipulation of system models using different abstractions has been a task allotted to the experienced modeller, who traverses the modelling life-cycle, from goals, through multiple abstract models into a concrete implementation. This process has hitherto remained internalised (*i.e.*, the modeller's experience), limiting the chances of its successful automation. Explicit "modelling" of this modelling process has been undertaken in the Software Engineering community and has resulted in a plethora of CASE tools. Similarly, explicit use of modelling expertise in engineering disciplines such as mechanical engineering and IC design has resulted in CAD/CAM and logic design tools.

Due to the continuing integration of software and hardware, there is a growing need for integrated modelling support. It is suggested that a rigorous modelling methodology will drastically enhance productivity, fully supporting the analysis, design or control system life-cycle. As in software development, introducing rigorous techniques will enhance (and make measurable) the correctness of system models and implementations. Possibly the most important consequence of a rigorous approach is the consistent re-use of model information. Strict limitations (constraints) are put on physical systems models as they have to represent reality. Using AI techniques, this vast knowledge can be employed to traverse the modelling life-cycle correctly. In particular, this a priori knowledge ensures correct re-use of models.

One of the main features of explicit model representation lies in the resulting simulations. At different abstraction levels, experiments can be performed on the model through simulation. This provides quantitative insight into the (analysed, designed, controlled) system. Through simulations, chances of propagating errors throughout the design are greatly reduced and insight into the dynamics of the system is enhanced.

In the design of a Generic Interactive System for MOdelling and Simulation, two complimentary aspects of modelling must be dealt with.
Firstly, the object-oriented *encapsulation* of knowledge in the form of abstract models. Different model types (algebraic, discrete-event, rules, data, ...) reflect the diversity of our knowledge as well as the multitude of formalisms used to represent and manipulate that knowledge. An appropriate encapsulation will allow a generic description of operations performed on the models.
Secondly, the process of *manipulating* model knowledge to satisfy certain goals. The processes of model building, model experimentation through simulation, model optimisation, ... transform the collection of models into a knowledge repository which grows through interaction with experts and users.

As model encapsulation and the modelling process are complimentary, it is important to give equal attention to the formalisation of both, as will be explained in further sections. In particular, the sub-processes of tactical and strategical design of simulation experiments are highlighted.

MODEL ENCAPSULATION

To consistently store and manipulate knowledge in the form of models, an object-oriented approach was taken. Each model is represented as an object, containing *all* relevant knowledge pertaining to that model. It was realised that a model, irrespective of the formalism (algebraic, discrete event, rules, ...) it is represented in, has a set of standard components :

- MODEL NAME: With the model type, the model name is used to uniquely identify the model in the Model data Base (MB). Depending on the level of instantiation, a model can be *concrete* (there exists an engine which can interpret –simulate– this model) or *abstract*. In the latter case, the model acts as a specification, for which different concrete implementations exist.
- MODEL TYPE: Determines the *meaning* of the model dynamics. The model type (or formalism) is associated – where possible– with an engine which performs *semantic mapping*: interpreting the following description (*e.g.*, algebraic, discrete-event, ...) of the model. As the generic model structure is independent of the model type, the modelling process can be generic. Model type dependent operations are performed through the model type field and the associated semantic mapping engine.
- MODEL COMMENTS: Describe the purpose, limits, references, ...for the model. These comments provide *explanations* when needed during the modelling/simulation/validation life-cycle. If *all* models (*i.e.*, system models as well as rule models prescribing the model choosing) are adequately documented, automated explanation generation becomes feasible.
- MODEL CONSTRAINTS: Any model of a system is only an accurate description of that system within a limited (albeit sometimes quite large) experimental range. That is, if experiments are performed both on the system and on the model (through simulation) within that *Experimental Frame*, both will deliver the same results. It is crucial to provide an Experimental Frame with every model to explicitly represent the model's limitations and hence its applicability range. The constraints are used intensively in the model choosing process. The term constraints was chosen as the Experimental Frame is generally expressed in terms of constraints on pertinent quantifiers (*e.g.*, distance must always be ≥ 0 [m]). Usually, constraints are chosen conservatively to ensure correct model re-use.
- MODEL PARAMETERS: Models can be parametrised, whereby different instantiations have different parameters. Through coupling, new models can be built with parameters which are (algebraic) combinations of the submodel parameters. Also, when connecting the appropriate output connections of an *optimiser* model to system model parameters, parameter optimisation can be performed.
- MODEL INTERFACE: The encapsulated model interacts with its environment through its interface, which consists of a set of connection nodes. The connection nodes (*e.g.*, input/output variables in the continuous model case) can be INPUT, OUTPUT, INPUT/OUTPUT or UNKNOWN. This reflects causality (in a timed system) or information flow direction (in an untimed system). A typical *specification* for a system model would consist of the components (name, type, comments, parameters and interface) mentioned above. This then allows for diverse *implementations* compatible with the specification (top-down design). Through matching of specifications with concrete models in the Model Base, a bottom up design is supported.
- INDEPENDENT VARIABLES: Most prominently, time in a time dependent system. When coupling different models, the independent variables may be the same in essence (*e.g.*, time), but carry different names. This problem of semantics vs. syntax is the core of multi-paradigm modelling.
- STATE VARIABLES: In a formalism which uses "state" to describe a system, this gives an exhaustive list of all pertinent variables (in terms of which the dynamics of the system is expressed).
- MODEL DYNAMICS: Used to be called "the model". Thanks to the model description components described above, it becomes possible to manipulate (*i.e.*, couple, choose, ...) models without explicitly knowing their dynamics. When quantitative insight is required, the dynamics must of course be interpreted. This is done through the model type field, which reveals the formalism used in the dynamics section. The type field also determines the engine (*i.e.*, simulator) needed to "run" the model.
- MODEL RULES: This section contains facts and rules regarding the model, allowing an automated choice between alternative models (given a particular goal). Typical facts are: this model is linear, this model has quadratic time complexity. A typical rule is: IF this model is simulated with appropriate initial conditions and the results exceed a certain threshold THEN this is not a meaningful model. Thus, model rules can give rise to a chain reaction of further enquiries (such as the running of a simulation). This is quite natural, as a human expert will often need to conduct additional experiments to get conclusive information about the nature of a system.

The *representation* of models has been formalised in a syntax named MSL : Model Specification Language. It only provides a framework for specifying models. The individual model formalisms (as indicated in the model type field) determine the syntax and semantics used in the dynamic section. The structure of MSL determines the structure of the (relational) Model Base (*i.e.*, the database schemas). Though the implementation uses a relational database, the form of the models and the operations on them support an object-oriented paradigm. Apart from being a basis for implementation, MSL provides a common notation for modellers to communicate in.

To resolve the problem of unequivocally expressing meaning, an *ontology* is used. The ontology stores the mapping between the syntax of entities in a given knowledge domain and their meaning. One useful ontology contains *units* information. Variables will *always* be given with their units which allows for dimensional analysis (as a means of model verification).

A partially completed MSL description acts as a model *specification*. Different alternative *implementations* can be constructed to satisfy the specification. If a model has to be chosen from different alternative implementations, a rule model will guide the decision process (thereby relying on information stored in the

models). The specification/implementation process is recursive: an implementation may itself be a lower level –more concrete– specification (transformational approach). Specification and implementation may be of different model types, thus descending from an *abstract* (*e.g.*, HGPSS) level to a more *concrete* level (*e.g.*, C++). If both are of the same type, the implementation is usually a *refinement*.

Up to now, only *atomic* models (no coupling structure) have been discussed. A coupled model is defined as a structure containing a set of sub-models (which may be of different types) and a (possibly directed) graph describing connections.

```
coupled model ::=
 <set of sub-models, connection graph>
```

If possible, such a coupled model can be *flattened* into one (or more) atomic models. This flattening is explained in more detail in the next section.

Recursively coupling models supports *hierarchical modelling*.

The coupling relationship between different abstractions is described in the *Abstraction Coupling Graph* (ACG), which records which couplings `Abstraction_A - Abstraction_B` are meaningful. Often, coupling between abstractions will require insertion of *transducer models* which convert one type of information (*e.g.*, discrete) into another (*e.g.*, continuous).

Possible transformations (as during the implementation process) between different abstractions are described in the *Abstraction Transformation Lattice* (ATL). A lattice is used as a result of the partial ordering (from abstract to more concrete) which can be attached to the set of all abstractions.

MODELLING PROCESS MODELLING

The *processes* of analysis or design of systems within the framework of the Generic Interactive Modelling and Simulation Environment, use static model information from the Model data Base. The process model of the modelling/simulation/optimisation process can itself be represented (or even specified, during the implementation of GISMOS) using a discrete event modelling formalism. The advantages of such an abstract representation are:

- It describes/prescribes/proscribes the process, giving insight and facilitating implementation.
- It provides a tangible, static means of studying the complex modelling process and of resolving ambiguities in the design of a modelling environment dynamics.
- Time is represented explicitly, hence simulation of the process model gives quantitative insight in the modelling process.
- Both control flow and data flow are made explicit, providing a powerful means of specification.

The generic process is represented in Figure 1. This process model is applicable to simulation, optimisation, control, ...problems (depending on the goal choice).

Refining this process model hierarchically reveals the following core processes:

- *Coupling* of sub-models: As mentioned before, a coupled model contains a set of sub-models as well as a connection graph. The coupling process involves:

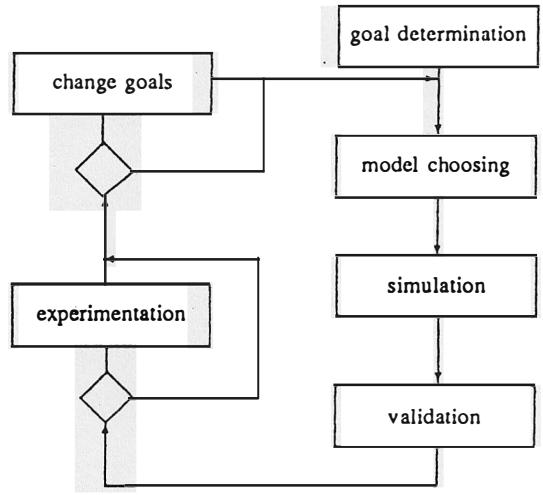

Figure 1: Generic Process Model

  - checking connection *consistency*: causality of coupling, type consistency, constraint consistency, etc.
  - where appropriate: insertion of *transducer* models.
  - transformation to a *common abstraction* CA.
  - flattening of the coupled model into one "flat" model of type CA. A model type is said to be "closed" under coupling if such flattening is possible (as is the case with algebraic models, C++ models, rule models, ...). If no common abstraction could be found, a simulation of the coupled model will comprise different interacting simulators.

  The coupling process provides a means of *hierarchical multi-paradigm modelling*, thereby encouraging model *re-use*.

- The *choosing of alternatives*: As many alternative implementations may correspond to a single specification (and goal), a choice often has to be made between alternatives available in the Model data Base. This choice is done on the basis of:
  - information embedded in the models (often requiring simulation of other models).
  - rules in the "alternative choosing" rule model.
- *Refining* of a specification: if the type of specification and implementation are identical, the process is called (stepwise) refinement.
- For each implementation step, correctness has to be *proven*: Only when the compliance of an implementation with its specification is proven, can *re-use* of models be truly meaningful.
- *Abstraction*: as the converse of implementation, abstraction goes from a specific model to a more generic one. Both abstraction and implementation traverse the Abstraction Transformation Lattice (in opposite directions).

A side effect of process modelling is the automatic derivation of a Finite State Automaton to be used as the basis for the design of the modelling environment Graphical User Interface.

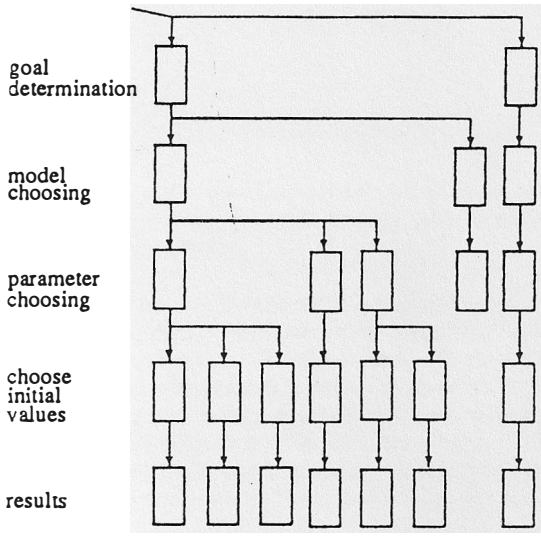

Figure 2: Virtual Product Life-cycle

MODELLING PROCESS HISTORY

The general process model is a complex structure, which (depending on the contents of decision models) becomes partially instantiated during the life-cycle of a model. Such an evolving model is termed Virtual Product, as it is a product description traversing different levels of abstraction (often ending in a concrete implementation: the product).

The Virtual Product Life-cycle (VPL) is a structure which records subsequent model life-cycles. In particular, it records all decisions made and their consequences (Figure 2).

As the VPL records *process history* (in particular, the relationship between alternative choices and their consequences), it can be used by decision (rule-based) models to learn from *past experience*. Hence, whereas the Model Base contains the "model memory", the VPL contains "decision memory". The VPL enables "learning" from past experience. Knowledge from the VPL may be collapsed (abstracted) into a rule model to be incorporated into the Model Base.

TACTICAL AND STRATEGICAL DESIGN OF SIMULATION EXPERIMENTS

The general modelling process as shown in Figure 1 is instantiated to build and simulate a model of a real system as well as to construct an "optimal" environment to manipulate such a system model. Building such an environment consists of choosing and instantiating the best "environment" model (*e.g.*, optimiser). Given the uniform way in which different types of models (algebraic, rule, optimiser, ...) are represented in the Model Base, the generic modelling process is applicable to the problem of choosing and utilising an environment model.

Obviously, the process of choosing the right environment is directed by knowledge embedded in system environment models and decision models in the Model Base. Undecidable problems are left to the user.

In the sequel, the design of simulation experiments (whereby a suitable system model to experiment on has already been found) is discussed. The following tasks must be performed:

- Analysis of user information about design goals and conditions.
- development and realisation of tactical design of simulation experiments.
- development and realisation of strategical design of simulation experiments.

Tactical design of simulation experiments ensures necessary *preciseness and reliability* of simulation runs. It solves following problems: sets initial conditions, determines length and number of simulation runs, analyzes the transient period and fixes the moment to start collection of (statistical) data, chooses a procedure to evaluate the model output.

Strategical design finds *optimal* values of the model parameters, which optimise the goal function. This function is formulated by the user on the basis of model output variables. The design depends on the concrete situation: the number of parameters to be tuned and their types, the dimension of a goal function, etc. Following situations, typical for simulations projects, are considered:

- With respect to the *number* of model parameters: few parameters; many parameters.
- With respect to the *nature* of model parameters: quantitative continuous parameters; quantitative discrete parameters; quantitative mixed (both continuous and discrete) parameters; qualitative parameters; mixed (both quantitative and qualitative) parameters.
- With respect to *restrictions* for possible meanings of model parameters: individual (interval) restrictions; correlated restrictions.
- With respect to the *dimension* of the goal function: scalar goal function; vectorial goal function.
- With respect to the *shape* of the goal function: uni-modal goal function; unknown shape of the goal function.

The design of an optimisation algorithm needs to take into account peculiarities of optimisation problems in simulation:

- The simulation model is *numerical* (non-analytical). Its optimisation asks for numerical optimisation methods.
- The goal function is evaluated from *simulation runs*. These runs are statistical by nature since they model random variables and phenomena. Their results also have to be treated as random ones. Hence, the goal function in simulation projects normally is a mean value of the quality function, which characterises the system under simulation. Evaluation of this mean value asks for numerous runs of the simulation model. As a result, each point in the optimisation procedure is expensive. Therefore, it is desirable to have an optimisation procedure with a minimum number of iteration points.
- High cost of the optimisation procedure necessitates determining its *termination conditions*. Usually these are formulated in terms of distances between iteration points and of the total number of iterations.

- The goal function may not satisfy *traditional requirements* of mathematical analysis, such as continuity and differentiability.

These peculiarities result in requirements for the optimisation algorithms. The optimisation algorithms used satisfy the above requirements. They are based on the following optimisation methods: random search (with linear and non-linear tactics), coordinate search, Hooke-Jeeves pattern search, steepest ascent, Box-Wilsson method, the genetic method. They also use principles such as two-stage optimisation (with pre-optimisation and fine optimisation), group screening and group optimisation.

IMPLEMENTATION

The current implementation builds on standards such as UNIX, X/Motif, a relational database, and C++. The different model types considered hitherto are:

- data model: contains experiment or simulation data, encapsulated appropriately to reveal its meaning.
- C++ model: contains C++ source code. This type of model is currently used to implement any part of a design which has no higher level abstract representation (thus, the environment will be bootstrapped, by implementing it in itself).
- optimiser model: contains encapsulated optimisers, enabling intelligent manipulation (*i.e.*, choosing) without explicit knowledge about the actual implementation.
- HGPSS model: the current systems modelled, simulated, optimised, ... are of discrete event nature. HGPSS [1], a hierarchical extension of GPSS [2] is used as the modelling language (and associated simulator).
- rule model: contains knowledge in the form of facts and rules, describing knowledge about models as well as decision (choosing) algorithms. The modelling language (and associated inference engine) used is cuprolog [3]. One of the virtues of cuprolog is its built-in constraint unification capability.

For each of these model types, an MSL description (and its meaning) are defined. From the MSL structure, database schemas are derived for model storage in the diamondBase [4] relational database.

A graphical user interface, based on the Finite State Automaton derived from the simple process model (Figure 1) is implemented using X/Motif.

FUTURE WORK

Future work will include:

- Further formalisation of the methodology. This includes correctness proofs as well as new algorithms (unification, coupling).
- Further implementation. In particular, the set of optimisers will be extended drastically (which implies an increase in optimiser model Experimental Frame insight).
- A hypertext based tutor, guiding the user through the undecidable parts of the process. Furthermore, the tutor teaches a novice about modelling and simulation and explains the use of the environment.

CONCLUSIONS

The unified approach to modelling and simulation is conceptually clean. Different abstractions are treated in a uniform fashion, not only lightening the mental effort required by the user, but also reducing the complexity of the implementation. Thanks to the integration of "software" (C++), "knowledge" (rule) and other model types, multi-paradigm design of systems (prominently hardware/software) becomes feasible. In particular, the "optimiser choosing" part of the modelling process benefits from the approach.

REFERENCES

F. Claeys 1992. "HGPSS: Object-Oriented Process Interaction Simulation." M.Sc. Thesis, University of Gent, Faculty of Applied Science, Belgium.

T.J. Schriber 1974. "Simulation Using GPSS." Wiley.

H. Tsuda 1992. "cu-Prolog for Constraint-Based Grammar." In Proceedings of Fifth Generation Computer Systems.

D. Platt; A. Davison; and K. Lentin 1993. "DiamondBase Version 0.2." Monash University Melbourne, Australia.