Laboratory of Economics and Management

Sant'Anna School of Advanced Studies

Piazza Martiri della Libertà, 33 - 56127 PISA (Italy)

Tel. +39-050-883-343 Fax +39-050-883-344

Email: lem@sssup.it Web Page: http://www.lem.sssup.it/

LEM

Working Paper Series

Laboratory for Simulation Develpment LSD

Marco Valente

LEM, Scuola Superiore S'Anna, Pisa, Italy and Universita' dell'Aquila, Italy

2008/12 June 2008

ISSN (online) 2284-0400

Laboratory for Simulation Develpment - LSD[*]

Marco Valente
LEM, Pisa
and
Università dell'Aquila
valente@ec.univaq.it

Abstract

LSD is one of many programming languages designed to develop agent-based models. LSD implements time-driven models expressed in formats equivalent to discrete systems of equations, where each equation computes the value of a generic instance of a variable at a generic time step. LSD models are therefore extremely parsimonious in terms of details that users must provide to the system. When a model has been described, the system automatically generates a working program implementing the model, endowed with a complete set of interfaces for any possible operation on the model.

The major feature of is that users can rely on an automatic scheduling system and on automatic retrieval of data required for the equations. Such features are particularly attractive in complex, multi-herarchical models. They permit even non-expert programmers to develop even relatively complex models with minimal training. The system's interfaces guarantee the complete control of the model at building, at run-time and at post-simulation analysis, facilitating debugging, revisions and detailed analysis of model results, which are useful properties especially when developing large models for ambitious projects.

The design of LSD is based on an "open architecture", so that LSD can be used to implement any type of model, including even-driven models and models based on customized data structures. The intrinsic modularity of LSD models make them easily scalable facilitating the development of highly complex models by demanding users. The underlining layer of C++, accessible by the users, allows the inclusions of external libraries or of complex data structures, besides an extreme speed and dimensions of the model.

This work reports on the major features of the design of LSD outlining its most prominent advantages for users of simulation models in research, particularly for agent-based simulations.

Keywords: Simulations models, programming languages

[*]The LSD project started in 1995 in IIASA, Laxembourg (Vienna), as part of the TED project led by Prof. G.Dosi. The project benefited initially by comments of other members of the project: F.Chiaromonte, W.Fontana, Y.Kaniovski, L.Marengo and G.Silverberg. Later, I continued to develop the system while studying for my PhD in Economics under the supervision of E.S.Andersen, who provided extremely valuable comments and suggestions. LSD has been developed following the needs encountered by myself, students and other users in our own research. Particularly demanding users, such as T.Ciarli and A.Lorentz who also assists me in teaching LSD, have to be therefore particularly acknowledged. Finally, all the students of SIME, a one week workshop on simulations with LSD organized for several years in Strasbourg by P.Llerena, have been extremely useful in testing and improving the system.

Introduction

Simulation models must obviously be implemented as computer programs, and there is, therefore, the need to identify a programming language. Although in principle any language can implement every possible computational structure, in practice different languages have different features, providing advantages and costs that may be relevant for different users and projects.

It is common to represent languages in a space represented by two hypothetical dimensions. A first dimension, let's call it *simplicity*, indicates how easy it is to develop a model with the language. For example, a language is simpler when it provides a large library of data structures and functions that users can choose and re-use, and the construction of a model requires minimal programming skills. Conversely, a language is more difficult when the user must learn a complex grammar and build every single piece of the model from very basic programming components.

A second dimension is the power of the language, represented by speed of execution, dimension of the resulting models, and flexibility of representation. A language is more powerful if it permits to build very fast simulations composed by large models, and allowing a large set of computational structures. Conversely, it is less powerful if it produces slow simulations, limits strongly the number of elements within a configuration, and makes hard to represent models different from a limited range of pre-determined computational structures.

The community of programmers represents languages along a trade-off between simplicity and power: simple to use languages have, in general, limited power, while powerful languages are very demanding in terms of skills and implementation time. LSD is an attempt to break this trade-off, offering the possibility to generate powerful simulation programs and requiring the programming skills implicit in expressing their own model.

The overall approach of LSD consists in keeping separate the simulation model from the simulation program implementing the model. The model, in LSD, consists in requiring the users to describe the model in terms close to the popular format of difference equations. From the set of equations the system automatically generates a powerful simulation program complete with an extensive set of graphical interfaces. The computational core of LSD models is extremely powerful, since it makes use of possibly the most efficient language, C++, therefore allowing the optimal exploitation of the computational power available. Moreover, the resulting program is automatically endowed with a large set of interfaces permitting to initialize, execute, investigate and, in general, exploit the model in order to satisfy any possible need that may require the use a simulation model. Users are therefore no required to provide any code concerning interfaces, files, memory management. Nor they are required to explicitly give instructions already implicitly contained in the equations; for example, the scheduling of computation for the equations is automatically generated at run time.

Besides being a powerful *and* easy to use language, LSD contrasts with other languages because of other features. Firstly, LSD models are "open" in the sense that users can constantly peek into the mechanisms of a simulation run observing every single event and accessing every detail of model at any time. Secondly, models implemented in LSD are extremely easy to revise, scale up and re-use. Thirdly, LSD permits to express very intuitively hierarchical models, composed by entities made of other entities generating many layers at different aggregate levels.

These three features make LSD models ideally suited for agent-based models aimed at research projects. Such models, in fact, pose particularly complex programming problems.

Such models are built, in fact, in order to study properties that normally cannot be assessed only observing a few data series. Modellers should be able to back their scientific claims by providing robust support based on the analysis of intra-simulation events. Failing to do so risks to strongly decrease the convincing power of simulation modelling as scientific tool. Also, as argued below, simulation models for research are intrinsically more uncertain in respect of normal programs developed for specific purposes. For this reason, it is very important to be able to trace an unexpected result to a bug in the code or to scientifically relevant properties of the model.

A second problem of simulation models for research consists in the need to frequently revise the model. By definition a researcher develops a model on the basis of conjectures and intuitions, without a perfectly clear understanding of the features that the model will need to include. For example, an element of the modelled system initially considered as irrelevant may turn out necessary, or a functional form may need to be heavily modified in the later stage of the research project. Consequently, the model must be constantly revised, modifying marginal details or the whole structure. The intrinsic modularity of LSD models makes extremely simple to edit a component of an existing version of the model minimising the effects to other parts of the model. For the same reason, a model in LSD can easily be upgraded adding new elements or new features without the need to adjust exiting ones. Also, it is very easy, and actually encouraged, to re-use portions of different models that will automatically be adapted by the system to new contexts.

Lastly, agent-based models make difficult to indexing the different copies of a given entity, because of the large sets of indexes required to identify entities at different levels of the model hierarchy. LSD offers an extremely simple system to solve the problems related to ensuring that the proper values are used in every computation.

LSD users need to know sufficient programming as necessary to express the equations of their model, while the system automatically generates all the rest of the required code. Since, in general, most of the models' code concerns a few simple logical-mathematical operations, LSD permits even poorly trained programmers to develop and exploit rather elaborated models. The use of the interfaces can be learned in few hours, and users need to concentrate on the limited set of programming structures directly relevant for the model content, ignoring completely core aspects of standard programming courses such as file and memory management, interfaces, etc. The scalability features of LSD implies that it is an ideal teaching tool for the logic of computation, allowing to ignore the difficulties programming languages.

However, LSD is by no means limited to be an educational tool. In fact, the features mentioned above make LSD uniquely qualified to develop large and highly complex models. In particular, LSD favors a gradual approach to modelling, adding few elements at a time, testing them, evaluating the scientific correctness, and adding further extensions. Using a standard programming language such approach, together with the uncertainty on future extensions, implies the reaching of a stage in which prior design choices and the complexity of the model prevent further changes. Actually, one is likely to discover that, somehow, a bug had been introduced in earlier versions, but there is no way to identify exactly the faulty lines or to fix the error without a complete rewriting of the model. LSD avoids these problems, permitting a total control of even extremely complex models. Such characteristic, together with its computational efficiency, make LSD very useful even for expert programmers.

This document aims at providing a brief overview of the main features of LSD. The first section provides a definition of a model as required by LSD. This is the only information that users must enter in the system to generate a simulation program and, as will be seen, it is a definition very close to the abstract representation of a model and far from an actual programming structure. The second section describes how the LSD system manages to turn the model definition into a simulation run. Such section briefly outlines the major features of the simulation engine of the system that analyses the user provided information and generates the implicit programming steps. The third section reviews some of the most relevant tools automatically provided by the system and allowing users to exploit the model. The concluding section summarizes the paper and comments on the limitation and future development of the system.

1 Elements of LSD models

The goal of LSD is to request modellers to provide all and only the information concerning exclusively the content of the model, and then the system automatically generates all the necessary technical code required by the simulation program, e.g. interfaces, file management, scheduler, etc. In so doing LSD proposes a sort of a normal form for simulations, defining all (and only) the elements affecting the model results. The elements of a model as devised by LSD are: objects, variables, functions and parameters[1].

1.1 Objects and model structure

LSD defines objects as containers of other elements of the model[2]. Objects should be considered as representations of entities of the real system represented by the model. The model structure is composed by the set of objects composing the model. Since objects can contain other objects, a model structure should be thought as a hierarchical tree made of objects, where each object has a unique container (parent object) and can contain many different types of objects, each expressed in multiple instances.

For example, a model may contain as top-objects entities called Country, containing three types of objects: Supply, Demand and Gov. Object Supply can then contain several instances of the same-type objects called Sector, in turn containing objects Firm, etc.

The role of objects consists in storing other elements, that is variables, functions and parameters, besides, possibly, other objects. Since every element must necessarily be stored in an object, the model structure determines how many instances of each variable, function or parameter will be present in the model. Multiplying the copies of an object the system automatically generates as many number of copies of the contained elements, since LSD forces each object of the same type to have the same structure.

The design of a model structure may be less than obvious, although a few rules of thumb generally lead quickly to a settled definition. In general, "larger" objects must be high in the hierarchy of a model and "small" ones should be lower. The role of the objects during a simulation run consists in directing the search of data required by the computation of variables (see section 2 on the Lsd model manager below). If the equation of variable X requires the value of, say, parameter α, and all the copies of this variable must use a common copy of this parameter, than α is likely to be located in an ascending object in respect of the object containing X. Conversely, if the equation of X requires to use several copies of α, than the sets of objects containing this parameter will be descending

[1]Since modellers can access directly the underlining C++ layer it is possible to extend a model with any programming structure that such language can express, including external libraries. In this document, however, we do not provide details on these features.

[2]LSD is not an object oriented language, in that the LSD objects do not have features such as inheritance. It should possibly be defined an *object-based* language.

(i.e. contained into) the object containig X. Lastly, if each X must use a specific copy of α, than the two elements will likely be stored in the same object.

LSD models stores all the required information independently, and only at run time the system assembles all elements and produces the necessary computation. This means that moving an element (say a variable) from one object to another does not necessarily affect the code for the equation of that variable, nor that of other variables using that element. Therefore, it is very easy to follow a gradual approach to model development, trying a given structure, observing the results, revising the structure and so on.

1.2 Variables, functions and parameters

Variable at the core elements of a LSD model, since modellers can express a computation exclusively by inserting code as the equation for a variable. They are defined a label and a piece of LSD-C++ code used to compute the value of the variable at each time step. The system ensures that each variable executes the code corresponding to the its equation once and only once, generating a value associated to the time step.

The procedure to write the code for a variable is extremely simple, at least for the vast majority of variables in models. In fact, the user is not requested to use indexes to specify the location of the values used in the equation, but simply refers to them by using their labels, much as the standard format of difference equations systems. For example, suppose that the equation for variable Q expressing the quantity produced by a firm is a function of the price p. The equation will be the LSD code equivalent to $Q = f(p)$ and will remain identical independently on whether the model defines the price as being contained in the objects for firms or, conversely, it is a market price contained in another object. In general, variables' code consists in an extremely simple and intuitive equivalent of the mathematical expression of a difference equation model, extended to include every legal computational structure like logical statements (e.g. IF-THEN-ELSE), cycles etc.

Variables can also contain statements overwriting model elements (e.g. to replace the value of a parameter), adding and removing objects, sorting objects, etc. The most frequently used computations can be expressed with a macro language so that coding equations for most of the cases can be easily learned in few hours even by non-expert programmers.

The only necessary requirement for the equation of a variable consists in providing as output one single value. At each time step that value will be associated to the variable. Obviously, the same code will be re-computed for each copy of that variable, where results will be produced because of the values used in the equations' computation.

Functions are essentially identical to variables, but are treated differently by the system. While variables are automatically updated by the system once and only once at each time step, functions are computed only (and always) when requested by other computations in the model. Therefore, while variables generates always a single value for each time step, functions may generate several values, or none, depending on how many times their value is requested.

Parameters are equivalent to variables whose equations consists in returning the value already stored.

1.3 Initialization and options

The initialization of a LSD model consists in the quantity of objects and in the initial values for variables, functions and parameters, if necessary.

The number of objects can be assigned either identically for each branch of the model or differently. Determining the number of objects correspondly determines also the number of copies of the elements contained in them. For example, consider a model with a structure composed by an object Market containing a set of objects Firm. Defining three markets generates three sets of Firm's, contained in each copy of Market, whose numbers can be set to different values. In so doing, it is also determined how many copies of the elements contained in Firm are present in the model.

Variables and functions require initial values in case they are requested with lagged values in at least one equation. For example, suppose that a variable is requested with lag 2 in one equation (i.e. the equation uses the value of the variable at two preceding time steps). In this case, when the simulation starts, at time step 1, there are no past values for this variable, and the user must provide the value for the variable at the fictitious time step -1. At step 2, equivalently, the variable will be used with the value at time 0. From step 3 onwards the system will use the values computed at earlier steps.

Functions also can be used with a lag (and therefore may require initial values). Only, the "lags" for functions concern times of activation, and not time steps.

Users need to provide also other options affecting the model's behaviour, such as the number of time steps for each simulation run, the "seed" for the pseudo-random series, the list of the elements whose values must be saved for post-simulation analysis. It is also possible to instruct the system to compute a sequence of independent simulation runs, each using a different pseudo-random seed, whose results will be saved in files. In this way it is possible to perform robustness tests, since all the files can be loaded into the post-simulation module and the results compared.

2 LSD simulation manager

The core feature of LSD consists in allowing the users to provide only a very minimal description of their model, while the system automatically assembles all available information and generates a complete simulation program. A simulation step can be thought as the sequence of equations computed according to an appropriate order and using, for their computation, the appropriate values. Obviously, it is the user that ultimately determines the operations in a simulation step, but the system allow to express these decisions in an extremely simplified and intuitive way, avoiding redundant and complex instructions.

Expressing a model using a difference equation model there are two major decisions to be made when the simulation step must be computed. Firstly, it is necessary to determine the order in which the equations are executed. Secondly, in general equations use data from other elements in the model present with many copies, and it is necessary to choose a specific copy for each equation computed. In a standard program the first problem is solved by requiring users to provide a complete schedule of the order of updating for the equations. The second by using an indexing of the elements. Both solutions require the user to give global instructions, indicating when each and every equation must be executed, and giving the precise indication on which elements to use for each equation.

The approach used by LSD consists in requiring users to provide only local instructions within an equation's code, and relying on the system to generate a global solution to both problems by assembling all the implicit information scattered in the equations' code and the structure of the model. For example, concerning the scheduling of the equations, the user only indicates (implicitly) if there is the need of a specific priority among a some of the equations. Similarly, the user can specify which data to use within an equation using searching rules, not giving the exact address of the required element. In both cases, the system adopts a default procedure by following the indications of the modeller, if they exists, and relying on the most obvious choice otherwise.

Such approach has two advantages. Firstly, minimize the amount of instructions required by the modeller. Secondly, makes the model easy to scale up and revise, since the same code will be automatically adapted to a revised model. The LSD simulation manager (LSM) is the system's module responsible to fill the users' supplied information with the implicit information contained in the model as a whole. In the rest of this section we give some detail on the mechanisms used by the system to solve the scheduling and data retrieval problems.

2.1 Automatic scheduler

Users describe equations as independent pieces of code resembling a system of a difference equation model. By independent is meant that, in the equations' code, there is no explicit instruction on the global order of execution of an equation. It is the system that ensures that the resulting simulation program will follow the implicit order as it can be reconstructed by the equations' code, on the basis of the lags' notation.

For example, consider a model made of only two variables, whose equations are $X_t = f(Y_{t-lag_y})$ and $Y_t = g(X_{t-lag_x})$, where $lag_{x,y}$ are the lags, that can take the values 0 or 1. There are three legal possibilities for the order of computation of these two variables, and an illegal one. The table below lists all the possible cases and the resulting implicit indication on the order of execution of the variables.

Eq. for X	**Eq. for Y**	**Order**
$X_t = f(Y_t)$	$Y_t = g(X_{t-1})$	Compute firstly Y and then X
$X_t = f(Y_{t-1})$	$Y_t = g(X_{t-1})$	Irrelevant order
$X_t = f(Y_{t-1})$	$Y_t = g(X_t)$	Compute firstly X and then Y
$X_t = f(Y_t)$	$Y_t = g(X_t)$	Illegal model, generates a dead-lock

Concerning the illegal model, this is due to the fact that LSD is a programming language, not a mathematical problem solver. Therefore, the equations are actual steps that must be performed by the processor, not conditions to satisfy. Therefore the system of equations $X_t = f(Y_t)$ and $Y_t = g(X_t)$ is a valid mathematical statement (i.e."find the values X and Y such as to satisfy the stated conditions") but an illegal computational structure, representing an infinitely looped chicken-egg problem (compute X after the computation of Y and compute Y after the computation of X).

Technically, LSD uses a stack system. At the beginning of a step, it start scanning all the variables of the model, trying to compute each of them, and then deciding, depending on the state of the variable and of the required values in its equation, whether to complete the computation or performing preliminary other tasks. For example, suppose that it tries to compute X in the model represented by the first row in the table above. If variable Y had been already computed at the current time step, the system supplies the resulting value without re-computing again its equation, and the equation for X can be safely completed. If, conversely, variable Y had not been computed as yet at the current time step, then the system suspends the computation for X, and executes the equation for Y immediately. Once Y is updated the execution of the equation for X continues from the point where it had been interrupted. In this latter example, when the scanning reaches Y the system will find that the variable had been already updated and does not re-compute it again.

In case the system finds a model like those described in the last row above, then the simulation is interrupted, and a error message it issued specifying the variables concerned.

The system works because LSD attaches to the variables' values the time step when it have been computed. For functions, instead, there is no such association, and therefore their equation is re-computed every time their value is requested in other equations. If a function is requested with a lag (its "past" value), the system returns the value from the previous computation.

2.2 Automatic data retriever

LSD equations normally do not specify where the elements required for their computation are stored, that is which object contains them. This may potentially generate ambiguities when there are many copies of the elements involved. For example, consider the equation $Q = f(p)$ in a model where Q is a variable defined in object Firm. The equation's code makes no reference to where p is located, nor which copy of p should be used if many exist in the model. Indeed, the same identical code may be used in different models where p could be located in Firm or Market, obviously generating the intended result.

For example, imagine that p is located in the same object Firm as Q. The equation's interpretation will therefore be: every Q must use the p stored in the same copy of Firm containing the computing Q. Consider, conversely, that p is located in object Market (parent of Firm), and that the model has three copies of Market. In this case, the interpretation of the equation becomes: for every Q uses the copy of p stored in the copy of Market containing the copy of Firm storing the computing Q.

In general the LSM adopts the following rule when searching the copies of the element to be used for the computation of a variable: use the copy of the element stored in the closest object to the object containing the computing variable.

2.3 Overruling LSD default

The automatic decisions taken by the system allow to express a model with minimal requirements of code and permits easy scalability. For example, in order to reverse completely the scheduling of operations a modeller can simply insert a lag in one equation. However, there are cases in which the modeller needs to perform operations different from the default behaviour. For example, a model may require that a variable of a firm in the first market access the price of the third market. In this case the default data retrieval would not work, since also the first market has a copy of the variable price, which, without specific instructions, would be used. Similarly, the automatic scheduler may need to be disabled in some special cases. For example, in certain conditions a variable may need to execute its equation twice at the same time step, replacing its result produced at the first computation.

In these, and many other cases, the user can overrule the system default behaviour using one of the macro commands specifically designed to perform the operations on the LSD models. A set of these commands concerns the most frequent cases in which users need to violate the default choices. Moreover, when really complex programming is required, the system can address directly the LSD internal representation and using C++ command to manipulate it. For example, LSD admits only real-valued variables, but a user can create customized C++ data structures and C++ routines operating on them, using the LSD variables only to trigger and control the customized operation.

An example of data structures external to LSD models and experimentally included in the LSD source code consists of lattices. These are windows composed by several square cells posed in a grid. Lattices are defined, initialized and modified with C++ routines that users can access within the code for one of the LSD equations, linking the model's state (e.g. a variable's value) to the external data (e.g. the color of a cell).

Another example is the possibility to turn an element from one type into another. For example a model may be defined as having initially a variable, which is then turn into a parameter (so that its equation is no more computed), and, possibly, later re-turned again into a variable.

Actually, the very development of LSD takes place when a model requires a violation of the default behaviour. In these cases a customized solution to the problem is implemented using the C++ layer. If the problem appears frequently, then a generalized solution is devised (if possible) and a new macro command is added.

3 LSD automatic features

The main advantage of using LSD is that though modellers needs to insert a minimal amount of information concerning the model, this is automatically endowed with a complete set of interfaces allowing the full exploitation of the model results, even for the needs of very large and complex models. This section reviews briefly the major tools embedded automatically in LSD simulation programs.

3.1 LMM and Lsd Model Programs

The LSD system is composed by two programs, used to manage different aspects of the model. The program used to initiate a model, organize several models, and every time the equations must be revised, is called Lsd Model Manager (LMM). This program allows the user to organize different models and contains a text editor specialized in dealing with the equations of LSD files. The equations of a model are automatically stored by LMM in a C++ source file and compiled to generate a LSD Model Program (LMP) along the source code of the system.

The compilation process entails the usual makefile that can be customized freely by users. Non-expert programmers can rely on the automatic options allowing the compilation of a LMP with a single click. Compilation errors are reported so that illegal code can be easily found and corrected.

While LMM is unique for each installation of LSD, every model generates its own LMP, containing its specific equations and a common set of interfaces, identical for all LSD models. Using the LMP's the users can define the model structure, declare the elements, initialize the values, running simulations, and, in general, every other operation but those affecting the equations of the model.

3.2 Macros for equations

Every equation is a separate group of lines forming chunks of independent code within a unique file. As said, the equations are written using LMM and are composed by keywords from a macro language specific for LSD models, besides including the whole C++. For example, the lines for the equation $X_t = Y_t + 3$ could be expressed as:

```
EQUATION(''X'')

RESULT(V(''Y'')+3)
```

In general, an equation consists of a few lines collecting values from the model and expressing a simple logical-mathematical operation returning a value. The equation, as in systems of equation mathematical models, must express the computation required by the generic copy of the variable at a generic time step of the simulation. Different copies will make use of different values, therefore returning, for the same computation, also different values.

LSD provides an extended set of commands expressing the most frequently used operations. The example above uses the most frequently used macro of all, `V(''Y'')`, which returns the value of the element called Y within the code of an equation. A similar macro is `VL(''Y'',1)` which expresses the equivalent of Y_{t-1}, that is, return the value of the past value of an element. Since in LSD the modeller can only write code within an equation, there are also commands to act on the model, for example adding, deleting or shifting objects (for example when sorting). The editor in LMM provides graphical scripts assisting in the use of the macros, requesting users to give model-specific information and then inserting the complete text of the requested macro command, so minimising the number of typing mistakes.

Given the possibility to ignore issues of data retrieving and of scheduling, the writing of the equations code poses in most of cases no problems even to non-programmers. In general, even complex models can be broken down to many variables, each having each a very simple code. An extensive manual and many example models, furthermore, provide blueprints for most operations usually necessary in the model.

3.3 Elements declarations

LMP's contain the compiled code for the variables' equations, and are used for remaining operation concerning the model. The definition of the model structure (i.e. labels of the objects, variables, etc.), as well as the initialization and other simulation options are created using a LMP and stored into files. Such files, called configuration file, contain all the information concerning a specific run: names and position of the model elements, initial values, number of steps, etc. Consequently, every simulation run can be always replicated by loading its configuration files into the same LMP.

The declaration of the model entities consists in the simple entering the text label and the nature of the element, generating both a list-based and a graphical representation of the model. Editing an existing configuration, such as moving one element in a different object or suppressing elements, are also allowed.

3.4 Initialization interfaces

As said, there are two classes of initial values to be defined in order to launch a simulation run: number of objects and values for parameters and lagged variables. The LSD programs automatically detect the number of values that users must insert and generate suitable interfaces where each entry cell is associated to an element to initialize. A multi-digit coding system ensures that the user is always aware of which copy of an element the initialization interface is dealing with. For example, a parameter contained in a third-layer of the hierarchy of a model (e.g. stored in object Firm, contained in Market, contained in Country), will be indicated with three digits for the copy of Country, of Market and of Firm respectively. This feature makes LSD models are particularly suited for agent-based models, and in general for micro-founded models, given that complicated multi-hierarchical models can be easily initialized.

A further feature of LSD is that it exploits the full power of available hardware resources. This means that it is easily available to test a model with a few dozens of elements, and then, by simply entering one number, generating a model with millions of elements. For such models manual initialization (entering the initial values one by one) is obviously unfeasible. A separate set of interfaces allow to insert values automatically, using a large set of initialization functions assigning automatically initial values to all the copies of an element (or only to a specified sub-set as, for example, every second copy). Finally, for very particular configurations requiring a specific set of values the system allows to upload initial values from a text file.

3.5 Run time analysis

Another major feature of LSD models is that users can inspect and interpret models at run time, using a variety of interfaces to observe results, spot particular conditions, and edit model states in any respect (but for revising equations' code). In this paragraph we review some of these possibilities.

A simulation can be run in several modes. Listing them in increasing level of details (and decreasing speed): batch mode (no graphics, no user access, results saved in file at the end of the simulation run); graphical, no detail, accessible (no message issued); few details (step completed); graphical results presented at run time. Apart from the first mode, the user can always interrupt a simulation run and switch to another simulation mode. Moreover, a simulation can be interrupted altogether and users can access the model in a "debug mode"[3].

The LSD debugging interfaces provide a complete report on the state of the model, for example showing all the values for every element of the model, the updating time for variables, number of objects, etc. The same interface permits to search specific elements, modify the value of one or a set of element (manually or automatically, as for initialization), force the computation of an equation, adding or removing objects. Moreover, the user can enter in the post-simulation analysis module (see below), then return to the debug-mode and continue the simulation.

The system can enter in debug mode by several means: clicking a button while observing a run; on an equation's code request (using a one-line macro command in the code, also passing textual and numerical messages); at a pre-determined time step; conditional on specific values of an element (e.g. when variable X assumes values below 100).

3.6 Post-simulation analysis

A LSD model program is endowed with a module specifically designed to deal with data produced by a simulation run. The module can manage data from a just-finished simulation (or an ongoing one); load data previously saved from past simulation, even many files at once; generate new data as elaboration of data present in the module; upload current states of the model.

The module is particularly suited for dealing with the massive amount of data that modern hardware is capable of producing. Actually, since large models produce far more data than that can be contained even in large amount of memory units, LSD models allow to select which data series from a model should be saved in a simulation run. Other data

[3]The name derived by the need for programmers to inspect the working of a program mainly to fix bugs. However, for scientific purposes such activities are frequently requested also to investigate how a given result is generated.

are maintained in memory only as long as required to complete the computations at the current time step, and then discarded to make room for newly generated values.

Even limiting the selection of values, the analysis of a model may require tens of thousands of series, which would be impossible to manage with a standard tools of lists boxes. The analysis module provides interfaces able to scan quickly all the available data set and select only the series satisfying a given criterion. For example, it is possible to select all series for variable X contained in the same objects where Y was set at a specified value.

Users can generate graphical plots (time series and cross-section), scatter plots and a few descriptive statistics. The graphical output is represented windows that can be turned into postscript files. Both graphs and statistics can be controlled in a variety of ways: selecting time intervals, setting different scaling, choosing colors, adding labels, etc. For more advanced analysis the module permits to export selected series as text files in a variety of formats (e.g. tab-separated or fixed columns, with or without headers, etc.)

3.7 Documentation

Descriptions of simulation models' content is always difficult due to the intrinsic interdependency of each element with a number of others. LMP's can be used to inspect any aspect of the model elements, showing the equation of a variable, providing the list of the elements used in the equation, or the list of the variables making use of the element. While this is useful for modellers, it requires the use of the LSD interfaces, and cannot be used to document the model to the general public, or for inclusion in a paper. For this purposes the system automatically generates complete documents containing all elements of the model, listed by the objects containing them, and reporting all relevant information for each element: textual documentation (if available), equations' code, initialization values, links to related elements. The so-called LSD reports can be formatted in HTML, providing hyperlinks for connections between elements, or formatted LaTeXtables, for inclusions in documents.

4 Conclusions

This work presents the major features of LSD, a language designed to easily produce powerful simulation models. LSD is based on the assumption that modellers should be requested to provide only model specific information: variables and the code required to compute their values. From this information the system is able to organize the different pieces of code into a coherent simulation cycle, automatically filling any remaining part required to actually perform a simulation.

This approach simplifies extremely the generation of simulation programs, since modellers can neglect the global structure of the model and focus on each individual variable. Moreover, since the system automatically assembles the sparse information into a program, LSD models can be very easily revised and extended.

Simulation programs generated with LSD are automatically endowed with a complete suite of interfaces allowing easily to inspect, initialize, run models, and manipulate the results. The interfaces allow also to manage huge models, for example permitting the initialization of parameters using a large and flexible set of initialization functions.

Though LSD provides advantage both for non programmers (because of its simplicity) and skilled modellers (because of its computational power with large models), still it has aspects that may decrease its appeal, at least apparently, for certain kind of users. LSD major limitation concerns its nature of a programming language. Therefore it cannot be used as problem solver; for example, it cannot be used to solve a system of equations or any other problem (although, obviously, it is possible to implement and computational routine). Also, since LSD is based on C++ it is possible to include in the code any external library compatible with this language.

Expert programmers approaching LSD for the first time are initially puzzled by the impossibility to impose explicitly a detailed control flow for the execution of variables' equations. Although this is an advantage of LSD, users can still implement a sort of `main(...)` cycle by generating it within one of the equations of the model. However, after some time using the system, even trained programmers generally begin to appreciate the redundancy of this operation, which generally poses a useless and rigid limitation to the extendibility of models.

Currently, the development of LSD has reached a mature stage in which only cosmetic and presentation issues are programmed, at least in the near future. Because of the need to maintain its multi-platform nature (LSD is distributed as freeware for MS Windows, Linux and Mac OS X platforms) the graphics make use of Tcl/Tk, one of the few languages that, at the time of earliest design of the project, guarantee the legal and technical requirements. Recently more efficient windowing systems have been developed, and the replacement of this aspect is being considered.

LSD installation currently contains a number of disparate example models, that user can study and copy for their needs. However, it would be useful to include a more formal and coherent list of LSD implementation of the most commonly used computational structure. Similarly, the current version of the system requires users to cut and past different pieces from existing models, namely the computational content and the elements' definition are stored into separate files. It would be useful to link these related aspects in order to facilitate the copying of relevant portions from models with a single operation.