Matthias Meyer and Klaus Hufschlag (2006)

A Generic Approach to an Object-Oriented Learning Classifier System Library

Journal of Artificial Societies and Social Simulation vol. 9, no. 3
<http://jasss.soc.surrey.ac.uk/9/3/9.html>

For information about citing this article, click here

Received: 31-Mar-2006 Accepted: 01-Apr-2006 Published: 30-Jun-2006

Abstract

Learning Classifier Systems (LCS) have gained popularity in the realm of social science simulation. However, when it comes to actually constructing a LCS for a particular modelling purpose, it seems that every researcher must currently "reinvent the wheel". Taking this situation as a starting point, the objective of this paper is to present the basic ideas behind a LCS library, which can relieve simulation researchers of some of the technical work and can provide a generic structure for modelling LCS. The library is based on a strictly object-oriented approach. This provides flexibility in the process of constructing a LCS for a specific modelling purpose and encourages experimentation with various different assumptions. The paper is supported with examples based on experience in using the library.

Keywords:
Learning Classifier System, Modularisation, Experimentation with Assumptions, JAVA

Introduction

1.1

Learning Classifier Systems (LCS) have recently become quite popular. This can also be observed in the realm of social science simulation. In particular, they are used as a tool for representing cognitive processes and modelling learning capabilities. Examples of the application of LCS in social science simulations are, to name a few: Marengo (1992) using a Holland-Classifier-System to model learning agents in an organization solving a collective decision problem, Vriend (1995) studying self-organization in markets or Welch, Reeves and Welch (1998) modelling auditor decision behaviour based on a classifier system.

1.2

However, this popularity contrasts with the ease and flexibility LCS can be used.[1] When it

comes to actually constructing a LCS for a particular modelling purpose, it seems that every researcher must currently "reinvent the wheel". Contrary to the fairly popular use of LCS for social science simulation models, no common technical basis has so far emerged. Given this situation, the aim of this paper is to present to a wider audience the ideas behind a generic object-oriented learning classifier library, which we developed for our own research.[2] We hope that other researchers find this library helpful as well, either for using the library to reduce programming work or for drawing on the ideas behind the library to structure and inspire their own work.

1.3

Such a library can offer several advantages for social science modelling. First of all, it reduces programming work. Repast, for example, relieves the researcher of programming scheduling mechanisms, data-collectors or displays. Similarly, a library for LCS should ideally relieve the modeller from implementing mechanisms of selection, crossover, mutation, and so on. As a result, one can expect researchers to be less concerned with the mere "technical work" of programming and thus they are able to focus more on social modelling. Secondly, it widens the range of possible users applying LCS methodology. Such a library considerably lowers the level of programming skills required by researchers. As a result, researchers with less advanced programming skills are also now able to construct a LCS for their specific modelling purposes. Thirdly, using a strictly object-oriented approach for the library provides high modelling flexibility. It allows for many concrete modelling options by providing elementary building blocks from which various different types of LCS can be constructed. This also reduces costs for experimentation in a particular simulation model. For example, it is easily possible to compare the results of a Holland-Classifier-System with the currently popular ZCS or XCS, because they can easily be exchanged. Finally, the library can enhance the transparency and comparability of LCS modelling, because the different LCS would be based on a common structure provided by the library. This would even be the case if, in the modelling process, most of the given methods were overwritten in the derived objects and only little use was made of ready-made functions. Provided the structure given by the library has not been changed, it would substantially aid the understanding a particular program. From a scientific perspective, such increased transparency and comparability are important features (Chang and Harrington, in press).

1.4

The remainder of the paper is organized in four sections. First, the basic ideas behind a strictly object-oriented approach to a LCS library are described (Section 2). On this basis, we derive and explain the basic structure of such a generic object-oriented LCS library by applying object-oriented analysis (Section 3). Then, it is illustrated how the library should be used, and our experiences in using the library for different research projects are described (Section 4). Finally, a short summary and conclusions are provided (Section 5).

Object-oriented Approach

2.1

Before presenting the library itself, the basic approach behind it should be discussed in some detail. The library is strictly modularized and makes extensive use of major aspects of the object-oriented programming (OOP) paradigm.

2.2

A 'classical' approach to supporting social simulation researchers in modelling a LCS would be a library that implements readymade LCS of specified types and provides these types for inclusion into programs. However, this would restrict users to only some types of LCS, which reduces the range of modelling potential. The classifier library we aim to provide should be more generic and not a toolset, but more a kind of "construction kit" with

ready-made building blocks. An object-oriented approach can provide this.

2.3

Object-oriented programming-styles are widely used for agent based modelling (see Meyer et al. 2003) and seem to be a near 'natural' approach. Not only does modelling agents-as-objects reflect reality for many social simulation researchers in an ontologically appropriate manner, it is also the control-flow in many multi-agent-models that becomes too complex for procedural architectures with functional orientation.[3] The possibility of constructing encapsulated entities by defining classes and methods enables complex control flows, but object-orientation goes beyond the basic concepts of classes, messages and information-hiding. Because they are intended to ease development and increase reusability, the concepts of *abstraction*, *inheritance* and *polymorphism* are of great importance (e.g. Janßen and Bundschuh 1993).

2.4

In this respect it should be noted, that the importance of a strict object orientation stems not only from a technical, but also from a methodological point of view. If a programming code represents theoretical models (Ostrom 1988), code changes mean theoretical development. In social science simulation, LCS often represent core assumptions about the behaviour of agents and they can be programmed using a classical functional design (e.g. Butz and Wilson 2002). However, there are different types of LCS that share some components, but not others. They represent theories that are similar, but not identical. For example, many LCS work with rules, using strings of binary or ternary alphabet to represent conditions and actions and to describe the state of the world, but they use different variables to measure the quality of rules. They share mechanisms for comparing states and conditions, but use different mechanisms of selection or reinforcement. The respective means of construction a LCS reflects the underlying theory — decomposing a LCS-program into exchangeable modules enables its successive theoretical development.

2.5

The concept of inheritance is of particular importance to our generic design. It allows a hierarchy of classes to be developed successively and adapted to different circumstances. For theories in the form of computer simulations, the concept of inheritance is a means of systematic differentiation. Models and their underlying assumptions can be developed gradually by the overriding of methods, e.g. moving from a simple reinforcement mechanism to a more complex one.[4] Additionally, the OOP concepts of abstraction and polymorphism facilitate the definition of a signature of common methods by means of which an entire set of classes can be addressed. Objects stemming from the same abstract parent class can, to some extent, be treated the same way by methods from other objects and can, therefore, be exchanged for one another. It becomes possible to test different assumptions within a common simulation-framework, merely by instantiating objects of alternative classes which have a common superclass.

2.6

As a result, experimentation and cumulative science can be fostered if it becomes possible to substitute components and thereby their assumptions, without having to replace large, interwoven parts of the model. This can all be encouraged by making consistent use of the concepts provided by object-oriented programming. The layout of the library is designed in this spirit and attempts to meet that claim with regard to constructing LCS. This is the topic of the next section.

Library Layout

3.1

In order to create a generic library, a consistent object-orientation and modularisation seem

to be important fundaments. Therefore, in a first step the major building blocks needed for a LCS have to be isolated. Then, object-orientated analysis leads to an abstract model which constitutes the framework for our library.

3.2

The starting point concerning the layout of the library is the following observation. The basic types of classifier systems that can be found in the literature have much in common. For example, the classical Holland-LCS, but also the newer ZCS or XCS work quite differently at the top level, but consist of nearly the same *elements* and use similar *mechanisms*.[5] In each case, a classifier system is, as the name suggests, a set of classifiers. Classifiers are rules with a condition component and an action component. Usually, for each classifier, further parameters, such as strength and accuracy, are stored. Mostly, the classifier's rule components are expressed as strings of binary alphabet (0, 1), sometimes extended by a wildcard symbol (#). In order to choose an action to match a current situation, the situation is encoded into a message, using the same alphabet (typically without a wildcard). By comparing the message with the condition components of classifiers, a subset of matching classifiers is selected. These become part of a more or less complex bidding competition, which is based on conformity with the given message and its parameters. The winner of this competition is selected deterministically or randomly. A typical way of choosing is the so-called roulette-wheel selection, a random mechanism with weighted probabilities, using the calculated bids as weights. The winner's action component can either be interpreted as a new message to be processed again or as an action to be performed. The results of an action are used for learning. Rewards are distributed and the classifiers' parameters are updated. Finally, the set of classifiers is modified by a genetic algorithm. At the top level there is again a considerable variety of processes for updating rule sets and types of genetic algorithms. However, they again employ common basic functions. These include mechanisms for selecting and manipulating subsets of the classifier system and functions like crossover and mutation to be performed on them (see Holland 1992).

3.3

This description can already be used to derive some implications for the class-layout of the library. At first sight, there seem to be three central classes: one for classifiers as representations of condition-action-rules with additional parameters, one for classifier systems as sets of classifiers and one for messages as descriptions for environment states. However, a closer look shows that messages and conditions must be comparable to each other. According to Holland's model, actions can become messages again and they, conditions and actions all have a great deal in common. Usually, all can be described simply by strings, i.e. as arrays of symbols with a given length and defined alphabet. However, this is not necessarily so. For example, conditions can be represented by tree structures as used in genetic programming (see Koza, Rice and Roughgarden 1992). Furthermore, with a little abstraction, symbols do not necessarily have to be characters — any object may serve as a symbol. Therefore, because we intend to model a generic library, we begin with the definition of symbols, languages (as extended alphabets) and abstract situations as basic classes of our library (see Figure 1).

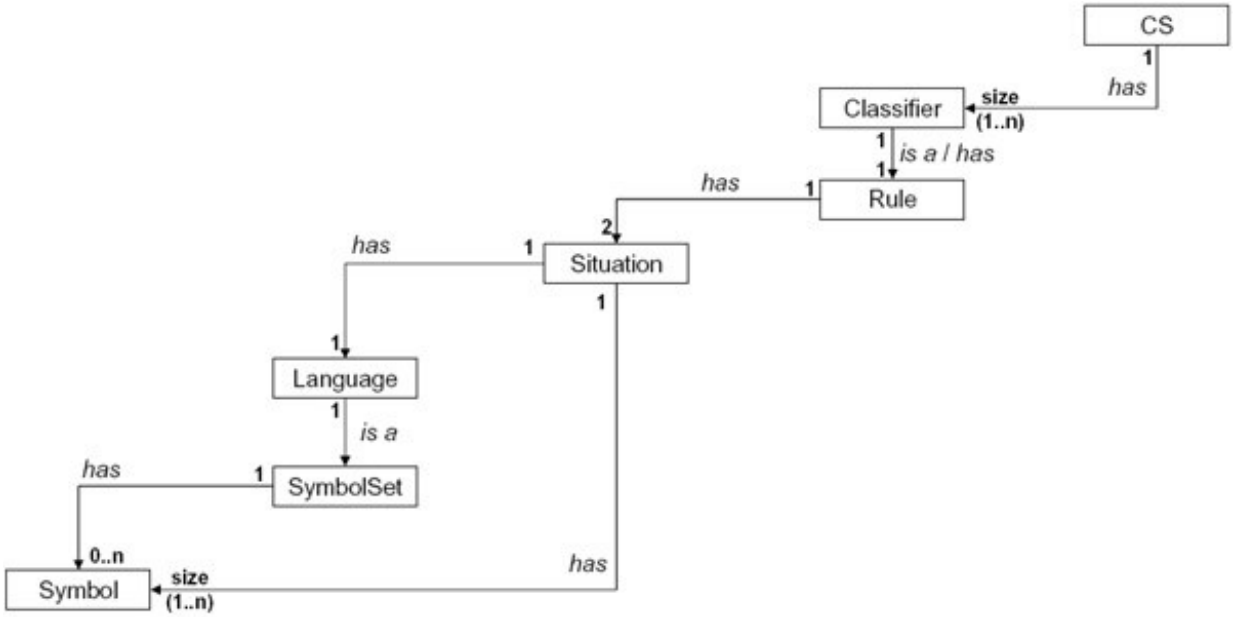

Figure 1. Simplified scheme of the library's main classes

3.4

The *Symbol*-class is the basic class of our library. Because every Symbol is thought to be instantiated only once, there is a mechanism for keeping its name-property unique. The class can be extended freely, for example, to add additional information or to associate it with specific methods. Symbols can be expressed in *SymbolSets* and *Languages*. The *SymbolSet*-class provides functionality for adding and removing single symbols or other sets of symbols from a set, for checking whether a symbol is contained in a set and for access, e.g. by delivering a randomly chosen or default object contained in the set. The *Language*-class is an extension of the *SymbolSet*, where the members of the set can be brought into hierarchical order. As a symbol subsuming '0' and '1', the typical '#'-wildcard can be defined that way. However, even more complex hierarchies and, by defining different languages, more than one hierarchy can be constructed.

3.5

A *Situation* can describe a state of the world, a condition or an action. As mentioned, it can be an array of symbols, a set of symbols or even a tree. Therefore, in the first step, it is defined abstractly and the typical string-like implementation is achieved as a subclass. Each *Situation*-Object must point to a *Language* that defines which *Symbols* are allowed and which relations of *Symbols* are to be used for interpreting the *Situation*-Object. During the LCS choice-process, messages and conditions are compared. In the genetic algorithm, conditions and actions are modified by crossover and mutation. Therefore, methods for comparing rules and for mutation and crossover are encompassed by the *Situation*-class.[6] Because the 'knowledge' of how to mutate or mate with another is located together with the *Situation*-objects, it becomes easy to change underlying procedures merely by overwriting the respective methods. Similarly, the classifier system implementing a genetic algorithm does not have to take these functions into account. The methods for crossover and mutation are transferred to the *Rule* and *Classifier* classes. A *Rule*-object encapsulates *Situations* for condition and action. A *Classifier*-object includes a *Rule*, but also stores additional parameters such as 'strength'. *Rule* and *Classifier*-classes must specify methods for crossover and mutation again, either just to call the methods of the encapsulated objects, or to add functionality, because, for example, the *Classifier*-classes need directions as to how to alter crossover parameters. Implementing two distinct classes for *Classifiers* and *Rules* is thought to be advantageous, as the structure of the condition-action rule is related to the problem domain, while the structure of the classifiers' parameters is associated with the learning-algorithm, especially the mechanisms for reinforcement and bidding of the chosen LCS-type.

3.6

Because a considerable amount of the functionality is already covered by the previously described classes, the main task of the *ClassifierSystem*-class remains that of filtering subsets itself, in calling methods on the subsets' members and in performing standard set-theory operations. Using these as major building-blocks, it takes only a few commands to implement a reinforcement-mechanism and a genetic algorithm for extending the *ClassifierSystem*-class into a complete LCS. The ready-made ZCS-implementation (*ZLCS*-class) to be found in the library is an example for this.

3.7

In this context it should also be pointed out, that he library currently implemented goes beyond the abstract level discussed above. It includes additional subclasses, implementing at least the standard variants of the abstract classes discussed above. Moreover, there are auxiliary classes included, like a roulette-wheel class, for programming convenience. In addition to the inherited functionality, the *ZLCS*-class shows how to construct a method for implementing the covering-mechanism, a method for processing rewards and finally "getters" and "setters" as accessory methods for the ZCS-Parameters.[7]

3.8

Finally it should be mentioned, that the library is based completely on the JAVA programming language. The main reasons are the elaborate and consistent object-orientation of JAVA and its broad diffusion within the scientific community. Together with technical platform independency, the latter opens the concept to a wide range of possible users.[8]

Using the Library

4.1

Making use of the library, it first has to be included in JAVA-projects using JAVA's standard import mechanism. Subsequently, two main stages can be identified in using its functions. The first is to choose and adapt the type of classifier system,[9] the second entails defining the way a single rule or classifier should look, in particular defining the structure of a classifier. The process for building a classifier system is described in Figure 2.

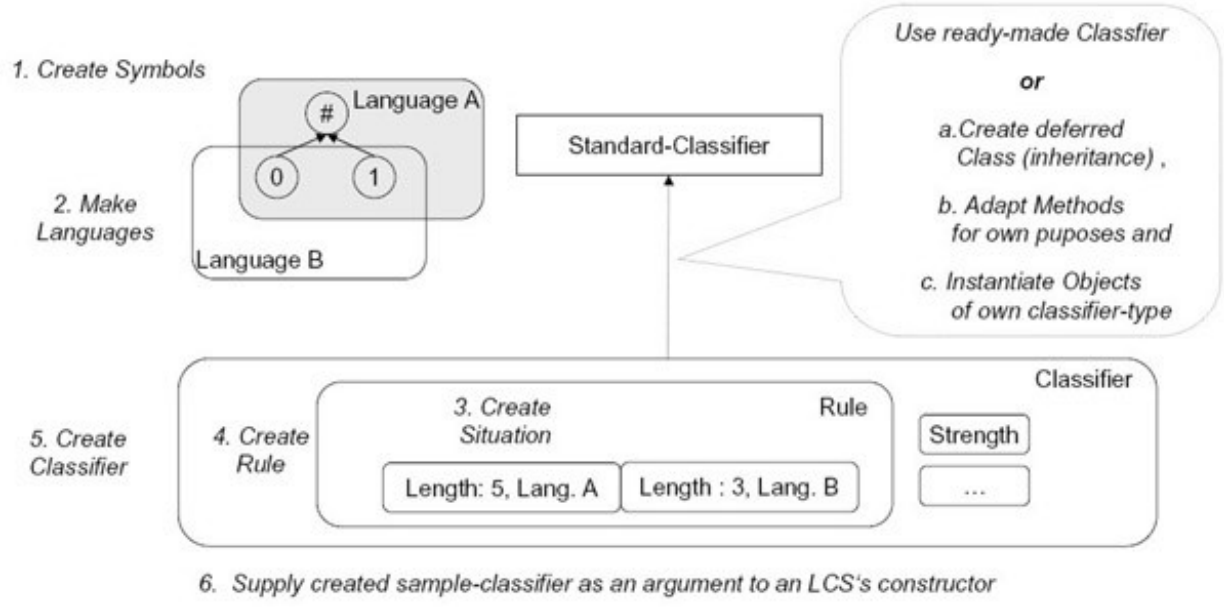

Figure 2. Steps for building a classifier system using the library

4.2

The definition of a classifier is straightforward and includes the following steps. The *Symbols* used in the simulation have to be defined (just calling constructors) and should be arranged in *Languages*. Then, sample *Situations* for the condition and action part of the rules have to be defined (usually by type, size and language) and for using them, a sample *Rule* must be constructed. The *Classifier*-object (e.g. of given *StandardClassifier*-class) is created by calling its constructor and pointing to the sample *Rule* defined earlier.[10] Finally, in order to set up a classifier system, the sample classifier defined has to be supplied as an argument to the classifier systems constructor. The sample can be multiplied automatically to attain the classifier system size (automatic randomization of situations is possible).

4.3

Concerning the application and applicability of the classifier library, we can already refer to some experiences. Currently, it has been used and tested in three different projects:

1. First, the library has been deployed in a replication-study of Marengo's model of "Coordination and organizational learning in the firm".[11] In this model, a simple Holland-like LCS (without Bucket Brigade) is used to represent individual learning capabilities and individual decision making. In this model, two decentralised decision makers and one central decision maker are placed in different organisational structures. Taking environmental dynamics as a second variable, the performance of different settings of this basic structure are compared with one another. An adaptation of the *StandardSituation*-class is used to match Marengo's method of describing conditions and actions and the *StandardClassifier*-class is extended to model the calculation of bids in the competition of rules. The *ClassifierSystem*-class is extended with functions to calculate parameters, but the genetic algorithm is implemented in a separate class. The replication project was temporary parallel to the library's development. Therefore, it did not use all currently available capabilities, but already demonstrated the general applicability of the library, with respect to a reduction of individual programming work.
2. In the second project entitled "Conceptual use of accounting information — a model-based approach"[12], the following research question is addressed: how can the rational-choice model of accounting information use be extended to include the conceptual use of information? This type of information use is highly relevant empirically, but this important aspect has so far been neglected in model-based research. Firstly, the analysis demonstrated the suitability of LCS as the basis for such an integrative, but still model-based approach. Then, as a prominent organization theory application of LCS, the Marengo model has been used as the starting point for the construction of a specific model of accounting information use. Several modifications to the basic structure of the model and the classifier system had to be made to allow for an analysis of the use of accounting information. These include agency and communication issues, as well as recent developments in classifier technology. The object-oriented approach of the classifier library provided the necessary flexibility to implement these modifications into the replication of the Marengo model.
3. Again, in another current project on "Information structures and bounded rational agents"[13], the library is being used for building LCS to represent individual learning in an organisational context. The project extends the team-theory approach of Marschak and Radner (1972) by assuming actors with bounded rationality. The abstract layout of the *Situation*-class enables modelling a more specific class derived from it (descendent class), which conforms to the specific team-theory representation of information. Subsequently, using Lindenberg's method of decreasing abstraction (see Lindenberg 1992), the actors' decision model is developed by successively shifting from a rational decision process to a truly bounded-rationality mechanism (see Radner 2000). The rational mechanism with full information can be modelled as a simple non-learning *ClassifierSystem* assuming a perfect rule set. The truly

bounded-rationality mechanism is represented by a LCS derived from that set, inheriting its basic functionality, but using non-maximizing (roulette wheel) choice, reinforcement-learning and a genetic algorithm. Thus, features of object-orientation such as inheritance and polymorphism have been used. The design of the library fostered an approach using the method of decreasing abstraction and allowed experimentation with various different assumptions relating to the particular LCS used.

4.4

Further applications are planned, but given our experiences and, because of the possible benefits of such a library for other researchers, we want to provide the generic classifier library as open source freeware to a wider public.[14] Hopefully, other researchers using LCS for social simulation models will also find it useful for reducing programming work and allowing for experimentation with different assumptions. Furthermore, the basic approach behind the library might be of use to others and give them some ideas for modelling their tailor-made LCS.

Summary

5.1

This paper presented the basic ideas behind a LCS library, which can relieve simulation researchers from some of the technical work and thus focus more on analysing social phenomena. The library is based on a generic and strictly object-oriented approach, going beyond the basic concepts of classes, messages and information hiding, by using the concepts of inheritance, abstraction and polymorphism in a consistent manner. This provides high flexibility in the process of constructing a LCS for a specific modelling purpose. In particular, it becomes easily possible to experiment with different assumptions in a common simulation framework. This has also been illustrated with reference to experience in using the library. Hopefully, other researchers find the library described in this paper useful, either to reduce programming work or for drawing on the ideas behind it to structure and inspire their own work.

Appendix: Code Examples

Example 1: Building a genetic algorithm using the library

Code	**Explanation**
Public void GA() {	Method-header, then random choice whether or not to run GA.
Random r = new Random();	
if (r.nextDouble() < getrho()) {	
ClassifierSystem pool = (ClassifierSystem) this.clone();	Make working-copy of this CS and generate new CS as set of victims (pool) to enable
ClassifierSystem victims = new ClassifierSystem();	

	random drawings (without putting back).
for (int lauf = 1; lauf <= getNumVictimsGA(); lauf++) {	Select given number of victims by roulette-wheel choice (here with inverted probabilities), remove them from working copy of CS and add to set of victims.
StandardClassifier victim = (StandardClassifier)	
pool.RoulettewheelChoice("getStrength", true);	
pool.remove(victim);	
victims.add(victim);}	
ClassifierSystem offsprings = new ClassifierSystem();	Generate new CS as set of offsprings
for (int lauf = 1; lauf <= getNumVictimsGA(); lauf++) {	Select given number of classifiers as offspring using (normal) roulette-wheel choice, remove them from working copy of CS. Strength of offspring is set to half, as classifiers get duplicated (according to ZCS-instruction). Add copy of offspring-classifiers to set of offspring-copies.
StandardClassifier offspring = (StandardClassifier)	
pool.RoulettewheelChoice("getStrength", false);	
pool.remove(offspring);	
offspring.setStrength((offspring.getStrength() / 2));	
offsprings.add((StandardClassifier) offspring.clone());}	
if (r.nextDouble() < chi) {	With given probability, start a pair wise mating-process of the classifiers in offspring set of offspring-copies.
offsprings = offsprings.crossover();}	
offsprings = offsprings.mutation(my);	Initiate mutation of the offspring-copies (after crossover) with given probability.
remove(victims);	Remove victims from original CS.
add(offsprings);}}	Add processed offspring to original CS.

Example 2: Defining symbols, languages, classifiers and LCS

Code	Explanation
Language lang1 = new Language("BasicLang");	Definition of language (lang1).
lang1.add(new Symbol("0")); lang1.add(new Symbol("1"));	Symbols are defined and added to Language lang1.

Language lang2 = new Language("ConditionLang");
lang2.add(lang1);
Symbol wild = new Symbol("#");
wild.addMember(lang1);
lang2.add(wild);
	2nd language (lang2) gets defined; all members of lang1 are added. In addition, a wildcard symbol gets defined and added to lang2.

StandardClassifier c1test = new StandardClassifier(new Rule(new StandardSituation(lang2, 10), new StandardSituation(lang1,10)), 1);	A classifier of type *StandardClassifier* is defined, nesting definitions of a Rule and of two *StandardSituations* with given languages and determined length.
ZLCS testsys = new ZLCS("Test", c1test, 400, true);	A new LCS of the ZLCS-type is instantiated, using the defined classifier as a sample for generating rules. The size of the LCS is 400 classifiers and the initial set is randomized.
testsys.setgamma(0);	Setting further parameters

Acknowledgements

The paper benefited from many discussions with Bernd-Oliver Heine. The authors also thank the participants of the European Society of Social Simulation annual conference 2005 in Koblenz for helpful discussions. Additionally, the remarks of two reviewers have

been helpful to improve the paper.

Notes

[1] In this context, one can also discuss the general suitability of LCS for the respective modelling purpose, which is beyond the scope of this paper. For a critical discussion of evolutionary algorithms as representations of social processes, see Chattoe (1998).

[2] This library will also be provided as open source. See footnote 14 for details.

[3] This complexity of control flows in computer simulation models led to the development of "Simula" programming language in 1967, as one of the origins of object-orientation (see Janßen and Bundschuh 1993). Moreover, because the control flow can become a matter of research in itself, classic procedural approaches in such cases fail to meet their objectives.

[4] For a methodological discussion see Lindenberg (1992), who suggests a stepwise development of assumptions as a "method of decreasing abstraction".

[5] For a detail description of these different classifier systems see Holland et al. (1987), Wilson (1994) and Butz and Wilson (2002). For a direct comparison, see Bull (2004).

[6] Mutation also can be used for altering environment states represented by a *Situation*-object.

[7] Example 1 in the Appendix shows how the ZLCS class uses inherited functions — especially selection and manipulation of sets of classifiers — to implement its genetic algorithm.

[8] Additional, as a practical side aspect, integration with Repast needs no further requisites.

[9] For the first step, the library itself comes with a ZCS implementation (ZLCS-class) that can be used as a ready-made LCS, but is mainly intended as an example of building a specific classifier system achieved by deriving it from the ClassifierSystem-class.

[10] The steps are also shown in Example 2 presented in the Appendix.

[11] See Marengo (1992). The replication is part of the SILC-Project, which is a cooperation of members of the chair of Controlling and Telecommunications at the Otto Beisheim School of Management in Vallendar (Bernd-Oliver Heine, Klaus Hufschlag and Matthias Meyer) and the Institute for IS Research of the University Koblenz-Landau (Klaus Troitzsch and Iris Lorscheid). The main objective of this project was to acquire knowledge of how to use LCS for modelling individual and organisational learning.

[12] This is a translation of the working title of the PhD project of Bernd-Oliver Heine (in German: Konzeptionelle Nutzung von Controllinginformationen — ein modelltheoretischer Ansatz").

[13] Translation of the working title of the PhD project of Klaus Hufschlag (in German: "Informationsstrukturen bei begrenzter Rationalität").

[14] A zip-File of the library's source code can be downloaded at http://www.openabm.org/model-archive/glcslib.

References

BULL, L (2004). Learning Classifier Systems: A Brief Introduction. In Bull, L (Ed.): *Applications of Learning Classifier Systems*. Berlin u.a.: Springer, pp. S. 3 -14.

BUTZ, M and WILSON, S W (2002). An algorithmic description of XCS. *Soft Computing*, Vol. 6, No. 3-4, pp. 144-153.

CHANG, M-H and HARRINGTON, J E (in press). Agent-Based Computational Economics. In Judd, K L and Tesfatsion, L (Ed.): *Agent-Based Models of Organisations: Handbook of Computational Economics II*. Amsterdam: North-Holland.

CHATTOE, E (1998). Just How (Un)realistic are Evolutionary Algorithms as Representations of Social Processes? *Journal of Artificial Societies and Social Simulation*, Vol. 1, No. 3.

HOLLAND, J H (1992). Genetic algorithms. *Scientific American*, Vol. 267, No. 1, pp. 44-50.

HOLLAND, J H, HOLYOAK, K J, NISBETT, R E and THAGARD, P R (1987). *Induction: processes of inference, learning, and discovery.* 2nd printing. Cambridge, Mass.: MIT Press.

JANßEN, H and BUNDSCHUH, M (1993). *Objektorientierte Software-Entwicklung.* München, Wien: Oldenbourg.

KOZA, J R, RICE, J P and ROUGHGARDEN, J (1992). Evolution of Food Foraging Strategies for the Caribbean Anolis Lizard Using Genetic Programming, Santa Fe Institute Working Paper 92-06-028, Santa Fe, NM.

LINDENBERG, S (1992). The Method of Decreasing Abstraction. In Coleman, J S and Fararo, T J (Ed.): *Rational choice theory: advocacy and critique*. Newsbury Park: Sage Publ., pp. 3-20.

MARENGO, L (1992). Coordination and organizational learning in the firm. *Journal of Evolutionary Economics*, Vol. 2, No. 4, pp. 313-326.

MARSCHAK, J and RADNER, R (1972). *Economic Theory of Teams.* New Haven, London: Yale University Press.

MEYER, D, KARATZOGLOU, A, LEISCH, F, BUCHTA, C and HORNIK, K (2003). A Simulation Framework for Heterogenous Agents. *Computational Economics*, Vol. 22, No., pp. S. 285-301.

OSTROM, T M (1988). Computer simulation: The third symbol system. *Journal of Experimental Social Psychology*, Vol. 24, No., pp. 381-392.

RADNER, R (2000). Costly and Bounded Rationality in Individual and Team Decision-Making. *Industrial and Corporate Change*, Vol. 9, No. 4, pp. 623-658.

VRIEND, N J (1995). Self-Organization of Markets: An Example of a Computational Approach. *Computational Economics*, Vol. 8, No. 3, pp. 205-231.

WELCH, O J, REEVES, T E and WELCH, S T (1998). Using a genetic algorithm-based classifier system for modeling auditor decision behavior in a fraud setting, Intelligent Systems in Accounting. *Finance & Management*, Vol. 7, No. 3, pp. 173 - 186.

WILSON, S W (1994). ZCS: A zeroth level classifier system. *Evolutionary Computation*, Vol. 2, No. 1, pp. S. 1-18.

Return to Contents of this issue