Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 1. pp. 127–134.

Advantages of a Multi-paradigm Programming Language in Agent-Based Model Definition

Richárd O. Legéndi[ab], Attila Szabó[ab]

[a]Eötvös Loránd University

[b]AITIA International, Inc.
e-mail:{rlegendi,aszabo}@aitia.ai

Abstract

This paper describes the advantages of the Functional Agent-Based Language for Simulations (Fables) compared to some of the most widely used toolkits of Agent-Based Modeling (ABM). Using Fables we implemented the *StupidModel*, a set of 16 simple ABM models those are designed to cover the most common features of ABMs. We also compared it to other published ABM platform implementations (Swarm, Repast and NetLogo) applying the Halstead code complexity metrics. In addition we studied some other aspects of these toolkits, e.g. the programming burden of auxiliary tasks of model implementation.

Keywords: agent-based modeling, programming languages, software metrics

MSC: 68T42

1. Introduction

The spread of the agent-based paradigm implied the development of software tools that supports ABM implementation. In the last years several ABM softwares has been developed to support the need of modelers (both programmers and social scientists). Various lists and comprehensions exist about these tools (like [1], [2] or [3]).

Albeit ABMs can be applied to many fields, model implementations consist only a well defined set of software elements: agent definition, agent communication, model dynamics, visualization (which is optional), and data collection (note that visualization alone can be a complex part of the model, especially in case of 3-dimensional surfaces). Some ABM toolkits utilize this by offering an ABM-specific language for model implementations (like NetLogo [7]); other tools offer an

application programming library (API) for existing programming languages (e.g. Repast J [6] and Mason [8] for Java).

The most desirable properties that ABM-specific languages offer are compact and also relatively simple source code, and a reduced risk of bugs. Ideally, its also easier to get experienced with a specific language than learning any programming paradigms (e.g. object-oriented programming) first.

The main advantage of an API is the freedom of the modeler to add arbitrary components to a model (e.g. components, that are not offered by the API). However, when the modeler is not an experienced programmer (but a social scientist, biologist, etc.), a general purpose programming language might be a burden to use.

The research reported herein compares the Functional Agent-Based Language for Simulation (Fables) [9] to two widely used ABM toolkits by applying Halstead software metrics. We used the three implementations of the same simple template model, the *StupidModel*, introduced by Railsback et al. [4].

This paper structured as follows. The next section introduces the Fables language briefly. Section 3 overviews Repast J and NetLogo. Section 4 describes how the platforms were compared. Section 5 provides the comparation results, and finally, we conclude the paper with a brief discussion.

2. Fables: Supporting the Methodology

Fables is a special purpose language for ABM and its integrated modeling environment that is intended to reduce programming skills required to create agent-based simulations. Its syntax is similiar to the mathematical formalism used in the publication in the subject which make it easier to read the code. The main design concept of Fables is to allow modelers to focus on modeling, and not on programming [9].

The language was designed to have no more language elements than required for a common model. It has defined structures for each specific part of the simulation: for the model and the agents (*object-oriented approach*), for the behaviour (*functional approach*) and for the model dynamics (initialization, stopping, agent interactions scheduling). In this way the Fables source code is very compact and straightforward in general.

A Fables model also splits the core model from the observer (i.e. the model definition from the visualization). Charts can be created with a point & click wizard in most of the cases, but custom datasources can be created through a graphical interface and/or scripting.

3. Evaluated Tools

In the next section we briefly introduce the examined tools, describing their main features, design concepts and goals.

3.1. Repast 3

The *Recursive Porous Agent Simulation Toolkit* (Repast, [6]) is an open source toolkit that was developed by Collier et al. at the University of Chicago. Repast 3 is an API available for different languages (like Python and .NET). In this paper we considered using the Java based implementation.

Repast has an abstract feature set focusing on modeling social behaviour, but is not limited to social simulation (a wide variety of applications is available ranging from evolutionary systems to market modeling and industrial analysis). Another advantage of Repast is its large and growing user community.

3.2. NetLogo

NetLogo [7] is a programming language from the Logo family. The primary design goal of NetLogo was to create an educational tool for academic students to help the learning of the basics of ABMs. It has an own, specific, easy to use programming language, as well as high-level structures and built-in functions for the common tasks for the model. It is one of the most popular platforms especially among the academic modelers.

The language greatly reduces the programming skills required to make simple simulation, hovewer it was clearly disegned for a specific model family: mobile agents acting in a two dimensional grid having local interaction patterns. These type of models can easily be implemented in NetLogo, but the platform is not limited to only them.

4. Comparing the Tools

The abovementioned agent-based model development platforms differ in various aspects, such as required programming skills, performance, or the length of code. In the following we describe the tools used for the comparison in this article.

4.1. Halstead Software Metrics

Software metrics are measures of some specific properties of a software or its component. These properties may be either derived from the source code only (*static code metrics*, like the total line of code or instruction path length) or influenced by some runtime behaviour as well (*dynamic code metrics*, like program loading time or number of bugs per line of code) . The research field of software metrics has an impressive literature from the '70s ([12], [13]): computer scientists and theoreticans tirelessly searching methods to define a measurement approach that can support software engineers with tasks such as making predictions about scheduling, costs, and upkeeping the reliability of a software product.

To compare the agent-based platforms we used the static Halstead software complexity metrics [11]. Fundamentally, the Halstead complexity metrics are based on

the number of operands and operators included in the source code. More specifically, the source code can be interpreted as a sequence of tokens. Tokens can be classified as operators and operands. Halstead derived code complexity metrics from the number of unique operators (n_1), operands (n_2), the total number of operators (N_1) and the total number of operands (N_2).

Length is the total number of all operands and operators, giving a sense how large is the code:

$$N = N_1 + N_2$$

Vocabulary is the total number of unique operands and operators, giving a sense how difficult is a statement (e.g. how much different functions are present in a source, the fewer is less complex):

$$n = n_1 + n_2$$

Volume uses the length and the vocabulary indicating the following information under consideration:

$$V = N * log_2(n)$$

A component with a high *volume* value should be refactored into smaller, simpler components. On the other hand, if *volume* of an analzyed component is scarce, the component may not have been given enough content or responsibility.

Additionally, empirical evidence suggests that these definitions can be used to estimate further quantitive measures as:

Difficulty describes the level of difficulty required to understand and maintain the code:

$$D = \frac{n_1}{2} * \frac{N_2}{n_2}$$

Effort suggests level of effort required to understand and implement the code:

$$E = D * V$$

The Halstead metrics are able to estimate other qualitative software attributes (such as the the required time to create the implementation or the number of potential bugs it may contain), but due to space limitations these aspects were omitted in this paper.

4.2. StupidModel

Railsback et al. defined a simple *template model* [4] for the following two major goals:

1. to provide template code for commonly used ABM features which can be used as a teaching tool for learning how to write ABM's

2. to compare ease of implementation of models in a variety of ABM platforms.

StupidModel 1-16 is an incremental series of models. In the first model bugs are moving on a grid randomly. Each following model extends the previous model by adding a common ABM feature to it (e.g. in *StupidModel* 2, constant bug growth is introduced). In the last model predators hunt the bugs, and bugs comsume the food available at their current cells; they also breed at a certain energy level, and die naturally afterwards.

Railsback et al. implemented this basic model in the most popular modeling environments including Swarm, Repast, Mason and NetLogo, and evaluated their capabilities through various metrics [5].

In this paper we made a contribution to this list: we implemented the *StupidModel* in Fables, and examined the design concepts, drawbacks and features of Fables, Repast and NetLogo.

5. Results

Table 1 contains the results for models 1, 8, and 16. As each model is derived incrementally from the previous one, the complexity of the models grows monotonously. Because of this, here we only provide results for *StupidModel* 1, 8, and 16: these results are enough to examine the trends of the code metrics in case of increasing code complexity.

The difference between an API and a specific programming language is clear from the results. Figure 1 shows that when using a specific language (NetLogo or Fables), all of the basic measures (that are used to calculate the complexity metrics) are far below the Repast J API-implementation's results in case of *StupidModel* 16. That is, model implementation in Repast requires a lot more effort than in the other two: the code of *StupidModel* 16 consists three times more tokens than any of the others, and its source code is distributed in five source files (for NetLogo or Fables, one file is enough).

According to the applied metrics, NetLogo requires the less programming effort (see Table 1 for results). It turned out that Fables use somewhat more operators and operands to describe a model (therefore the Halstead metric values are higher too). However, the number of unique operators show an interesting trend. While the Repast and NetLogo *StupidModel* 16 implementations consists significantly more uniqe operators than in *StupidModel* 1 (41 and 25, and 28 and 12 respectively), there's no significant change in case of the Fables implementations (29 and 26). It

Table 1: Halstead metrics results for Repast J 3.1, NetLogo, and Fables inplementations of StupidModel 1, 8, and 16.

	StupidModel1			StupidModel8			StupidModel16		
Metric	RP	NL	F	RP	NL	F	RP	NL	F
lines	82	29	32	257	82	68	541	177	124
files	2	1	1	3	1	1	5	1	1
funcs	8	3	9	33	6	14	64	9	26
calls	36	7	17	89	30	34	225	83	64
operators	81	32	63	245	77	132	519	198	266
operands	173	55	107	546	138	203	1135	375	425
uniqe ops	25	12	26	27	22	29	41	28	29
uniqe oprs	82	23	39	214	55	71	390	116	108
Length	254	87	170	791	215	335	1654	573	691
Vocabulary	107	35	65	241	77	100	431	144	137
Volume	1712	446	1024	6259	1347	2226	14475	4108	4905
Difficulty	26	14	36	34	28	42	60	45	57
Effort	45158	6403	36516	215588	37187	92272	863585	185939	279865
Length'	637	147	328	1785	416	578	3577	930	870

RP: Repast results, NL: NetLogo results, F: Fables results

means that when someone can implement the most simple model in Fables, she can also implement a relatively complex model using the same restricted words (or *"instruction set"*) – which is an important advantage of the language.

6. Conclusions

We compared the Fables language to the widely used agent-based model definition tools Repast 3 and NetLogo, and also measured the imlementation burden related to these software. We argue that ABM-specific languages are the future for agent-based modeling as they require substantially less implementation effort than APIs for existing general purpose programming languages.

We found that modeling in NetLogo - which is a widely used ABM platform in education nowadays - results shorter code compared to Fables. It was also revealed that a simple Fables model consists the same operators as a complex one. This is an important feature, which makes Fables ideal for non-programmers, and for eductional purposes.

7. Acknowledgements

The work reported herein benefited from the partial support of the Hungarian Government via the TAMOP project (grant TAMOP-4.2.1/B-09/1/KMR-2010-0003), which is gratefully acknowledged.

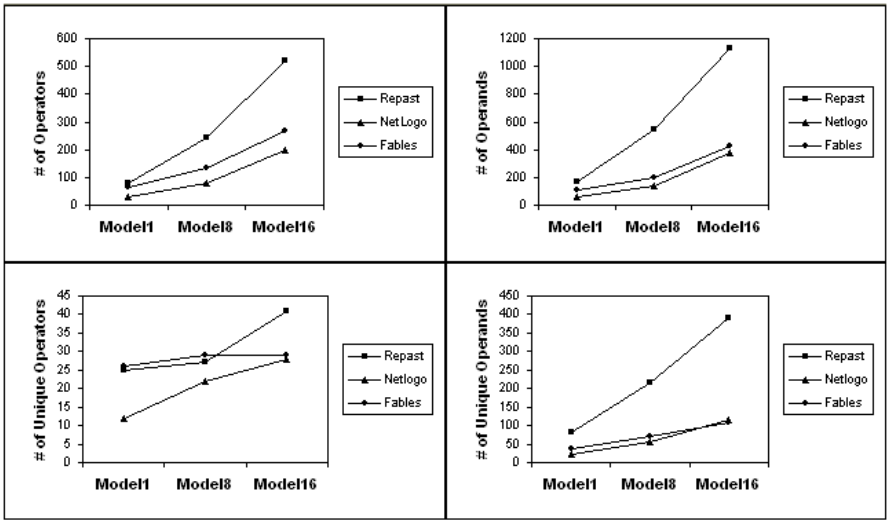

Figure 1: Change in the basic metrics: number of operators (upper left figure), number of operands (upper right figure), number of unique operators (lower left figure), and number of unique operands (lower right figure).

References

[1] Comparison of agent-based modeling software
`http://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software`

[2] Swarm Development Group Wiki, *Tools for Agent-Based Modelling*
`http://www.swarm.org/wiki/Tools_for_Agent-Based_Modelling`

[3] Leigh Tesfatsion, General Software and Toolkits Agent-Based Computational Economics and Complex Adaptive Systems
`http://econ2.econ.iastate.edu/tesfatsi/acecode.htm`

[4] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson, Stupid-Model and Extensions: A Template and Teaching Tool for Agent-based Modeling Platforms (2005).

[5] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson, Agent-based Simulation Platforms: Review and Development Recommendations. *Simulation* Vol. 82, No. 9 (2006), 609–623.

[6] North, M.J., Collier, N.T., Vos, J.R., Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit, *ACM Transactions on Modeling and Computer Simulation* Vol. 16, Issue 1, pp. 1–25, ACM, New York, USA (2006).

[7] Wilensky, U., NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL (1999).
`http://ccl.northwestern.edu/netlogo/`

[8] Luke, S., Balan, G. C., Panait, L., Cioffi-Revilla, C., and Paus, S. MASON: A Java Multi-Agent Simulation Library. Proceedings of the Agent 2003 Conference. (2003).

[9] Legéndi, R., Gulyás, L., Bocsi, R., and Máhr, T., Modeling Autonomous Adaptive Agents with Functional Language for Simulations. *Lecture Notes In Artificial Intelligence*, Vol. 5816., 449–460 (2009).

[10] Cem Kaner, Walter P. Bond, Software Engineering Metrics: What Do They Measure and How Do We Know?, *Metrics* (2004).

[11] Halstead, M. Elements of Software Science, North Holland, Amsterdam (1977).

[12] Wolverton, R. W., The Cost of Developing Large-Scale Software. *IEEE Trans. Computers C-23*, Vol. 6 (1974), 615–636.

[13] Perlis A., F. Sayward, and M. Shaw Software Metrics: An Analysis and Evaluation. *MIT Press* (1981).

Richárd O. Legéndi
Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest 1117, Hungary,
AITIA International, Inc., Czetz János u. 48-50., Budapest 1039, Hungary

Attila Szabó
Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest 1117, Hungary,
AITIA International, Inc., Czetz János u. 48-50., Budapest 1039, Hungary