

## AGENT-BASED META-MODELS

M.T. PARKER,\* Argonne National Laboratory, Argonne, IL,  
and Independent Consultant, East Hampton, NY  
M.J. NORTH, Argonne National Laboratory, Argonne, IL,  
and The University of Chicago, Chicago, IL  
N.T. COLLIER, Argonne National Laboratory, Argonne, IL,  
and PantaRei Corp., Cambridge, MA  
T.R. HOWE, Argonne National Laboratory, Argonne, IL  
J.R. VOS, Argonne National Laboratory, Argonne, IL,  
and the University of Illinois at Urbana-Champaign, Urbana, IL

### ABSTRACT

The agent-based modeling and simulation (ABMS) community has long recognized a need for concise, complete, and implementation-neutral representations of agent-models, and for modeling tools that do not require significant computer programming experience. We discuss earlier efforts to address these needs, arguing that proposed representations were typically too high-level and did not cover behavior. It may be that these weaknesses were insurmountable at the time—and that it is only now, with the availability of relatively mature Domain Specific Languages (DSLs) and Model Driven Software Development (MDSO) tools that these needs may finally be met. We justify this claim by identifying significant issues modelers face in using General Purpose Languages (GPLs) for agent-based models and how these issues might be overcome by using DSLs. We describe the specific tools we intend to employ in that effort and how we plan to use those tools, and we propose a general meta-model for ABMS.

**Keywords:** Agent-based modeling and simulation, domain specific languages, model driven software development

### INTRODUCTION

Agent-based modeling often demands sophisticated computer programming skills, limiting adoption of this revolutionary technique. While high-level mathematics is a core training requirement for physical, life, and social sciences, programming rarely is—effectively denying the power of agent-based modeling to a large swath of researchers and arguably biasing explanations toward traditional equation-based approaches. In addition, while significant efforts have been made to develop frameworks that leverage programming resources and provide model exploration tools to non-programmers (Parker 2000; Inchiosa and Parker 2002), the models themselves typically have been expressed in general purpose languages such as Java, which as we argue below, are neither transparent nor particularly expressive under many usages. These issues, coupled with a lack of consistent representations, have made it quite difficult to share models and to expose them to outside review. The authors and others have long sought solutions to these issues.

---

\* *Corresponding author address:* Miles T. Parker, 24 Gann Road, East Hampton, NY 11937; email: milesparker@gmail.com.

For example, Gulyás et al. discussed the potential of XML (e.g., Multi-Agent Modelling Language or MaML) as an agent-representation language as early as 1999. In the same period, Parker argued that strict compositions of hierarchical collections (“scapes”) with high-level spatial and execution model abstractions would facilitate the use of a declarative approach to model design in general (2000). More recently, the Repast team has demonstrated tools that allow extensive portions of model definitions to be made within XML (North et al. 2005). These approaches provide more transparent model representations, but there are typically still large gaps in what can be represented at such a high level. In particular, the proposed ABMS schemas typically have focused on structural issues and have left behavioral issues aside. As such, these schemas have not yet provided an effective and generic way to completely specify agent-based models. While one could argue that part of this weakness stems from the declarative nature of most of the proposals and that the difficulties encountered are largely representational, there may be other issues at hand.

## DOMAIN-SPECIFIC LANGUAGES AND META-MODELS

For many purposes, agents are naturally implemented as objects. Fortunately, object-oriented (OO) imperative languages are readily available to implement these objects. However, there are two important issues with this approach.

First, OO languages are general-purpose by design and so, to accommodate the universe of possible uses, carry complex syntax and semantics, very extensive Application Programming Interfaces (APIs), and idiomatic (but very general) usage patterns. Requiring such general and extensive knowledge is like requiring a person who simply wants to travel from Philadelphia to Boston to memorize the commercial route-map for the entire country and to become conversant with the maintenance procedures of the Boeing 737 to do so. Conversely, GPLs lack articulate idioms that fit a specific usage or context.<sup>1</sup> These kinds of idioms—e.g., shared agreement on the context of a particular communication and a “way of speaking” that uses this agreement to shape powerful abstractions and analogies—are what turn the symbols and structures of human language into artifacts of expressivity and beauty.<sup>2</sup> Doing without these idioms is like requiring text and instant messengers to use complete grammatical sentences with no abbreviations; or it is like requiring that the works of William Shakespeare be edited to replace any archaic language with contemporary “equivalents.”

Second, mainstream OO languages are compiled and static. We can distinguish these from “scripting” languages such as Javascript, Python, and Perl, which are interpreted and dynamic.<sup>3</sup> At the risk of making broad generalizations about controversial and complex issues, it can be argued that there are good reasons for these language design choices: they can encourage

- 
- 1 For an excellent example of how modeling and DSLs can address this issue by using a simple state machine language, see Voelter’s (2006) article.
  - 2 We leave discussion of what idioms may be most appropriate and powerful for ABMSs to a later paper.
  - 3 Compiled languages convert language-level code to machine code in one step (i.e., at “compile-time”). Interpreted languages do so while a program is actually executing (i.e., at “runtime”). Java is thought of as a compiled language because its instructions are written in Java “byte-code,” a platform-neutral machine language for the Java runtime environment. Under certain circumstances, Java can also be compiled at runtime. We are ignoring many subtleties and caveats here.

code quality; have a dramatic effect on performance (e.g., orders of magnitude); prevent significant security issues; and aid modern development environment techniques such as code completion and refactoring, and so on. But compiled languages also have a strong net effect of forcing design decisions into build-time (i.e., they prevent users from making significant changes in structure, as opposed to state and behavior).

Note that it is frequently possible to allow design decisions to be made at runtime, at least when we know at compile-time what the set of potential design decisions is. But that almost always involves sophisticated architecture and significant, sometimes dramatic, runtime slowdowns. As a simple example, suppose we want to allow model developers to provide arbitrary properties to agents at runtime. We could give a pre-defined agent type a collection of properties, but any time we wanted to access those properties, we would have to do so indirectly, e.g., through a map lookup.<sup>4</sup> We would introduce significant complexity and performance costs, and would lose language support for types, behaviors, and so on. In effect, we would almost be designing our own language, but a clumsy, inefficient, and opaque one.

Until now, there have been three general approaches to these issues. One approach is to simply employ a dynamic, interpreted scripting language and accept the costs of doing so, as discussed above. For example, this strategy was employed successfully in NetLogo (Wilensky 1998). The second is to employ techniques such as Java reflection that allow users to discover and update object state arbitrarily at runtime. While this approach allows a great deal of flexibility, it again imposes a severe performance penalty without solving fundamental issues. The third approach is to use code-generation techniques to compile and load custom agent designs. This approach is used successfully in the core Repast Symphony (Repast S) environment to support the use of Java annotations in specifying agent state “watchers.” The Repast for Python Scripting (North et al. 2006a) environment had advantages of the first and third approaches, but was of necessity a focused solution designed to support beginning developers. While code generation can provide good performance, it is typically inflexible and very difficult to implement and maintain. As such, it is best left to well-defined key subsystems. But the basic code-generation approach is sound, as the next discussion demonstrates.

Many see an elegant and general solution to these issues in the employment of Domain Specific Languages (DSLs) that capture idiomatic expressivity and implied context. In the past, the development of special purpose languages has been limited since the path to their development requires very specialized knowledge and substantial effort. But recently, tools have emerged offering support for such languages at a deep level and allowing them to be defined and implemented in a relatively straightforward and consistent manner. These tools, which partially fulfill the generative programming vision and are variously referred to as intentional programming tools, software factories, meta-programming tools, or what Martin Fowler aptly terms “Language Workbenches” (LWs), may be exactly the tools we need to build our agent-based modeling language.

Toward this end, one of us explored purpose-built LWs but theoretical and practical evaluation of these tools was halted when it was realized that it could not be predicted when these tools would be mature or even generally available. Later, the Eclipse project’s Eclipse Modeling Framework (EMF) was recognized as a candidate for this effort. EMF seemed to

---

4 In fact, this was considered in the earliest designs for Ascape, circa 1997.

support at least the sub-problem of modeling such a language's structure<sup>5</sup> and provides robust code-generation facilities to support it. For those unfamiliar with code-generation and other techniques mentioned below, the next section provides a brief explanation. Initially, experimentation focused on the Java Emitter Template (JET) toolset for the definition of code-generation mechanisms. JET is essentially a version of Java Server Pages modified for Java template use, and while this approach seemed workable, it might not provide adequately transparent, maintainable, or modular support for template definitions. Fortunately, a different toolset, Open Architecture Ware (oAW), has more recently joined Eclipse's official project. oAW provides a rich, problem-specific "template" language for code-generation called "Xpand2," a powerful constraints language called "Check," a model transformation language called "Xtend," and other supporting tools, including validation rules through the "Recipe Framework," and a workflow engine to tie the entire process together. More serendipitously, oAW can be extended with a language for defining DSLs using high-level Backus-Naur form (BNF) notation and can use this definition easily to build the kinds of supporting features that contemporary developers are beginning to demand, such as live syntax checking, syntax highlighting, and code completion. It is this toolset that we are now actively employing as we develop our new MDS/DSL system, which we describe in outline in the next section.

## THE REPAST SIMPHONY "SCORE" ENVIRONMENT

Our pilot effort<sup>6</sup>, dubbed "Score,"<sup>7</sup> utilizes EMF and oAW to provide the support we need for a complete ABMS environment and language. We now describe the current design in detail.

Figure 1 depicts the tool-chain we currently envision. Please note that we are describing how we will use the meta-model before we describe the meta-model itself, since it may inform readers about why the meta-model looks the way it does.

First, note that the model makes use of the Repast S code-base (ROAD [Repast Organization for Architecture and Design] 2006); that is, the set of APIs that a model developer can use to create and run a complete, Java-coded Repast S model. This is important because it ensures that there are no dependencies of the essential parts of the Repast S code base on the EMF or Eclipse systems. Next, we have the Score meta-model, an instantiation of EMF's Ecore meta-model that encapsulates everything we need to know about a model. For example, a model developer may specify a type of agent that has a happiness attribute and moves upon a 2-dimensional grid. Next, agent modelers design models for their particular problem space using EMF tools, such as the basic tree editor depicted in Figure 4; purpose-built graphical tools designed in the Eclipse Graphical Modeling Framework; plain XML; or a DSL implemented by using xText.

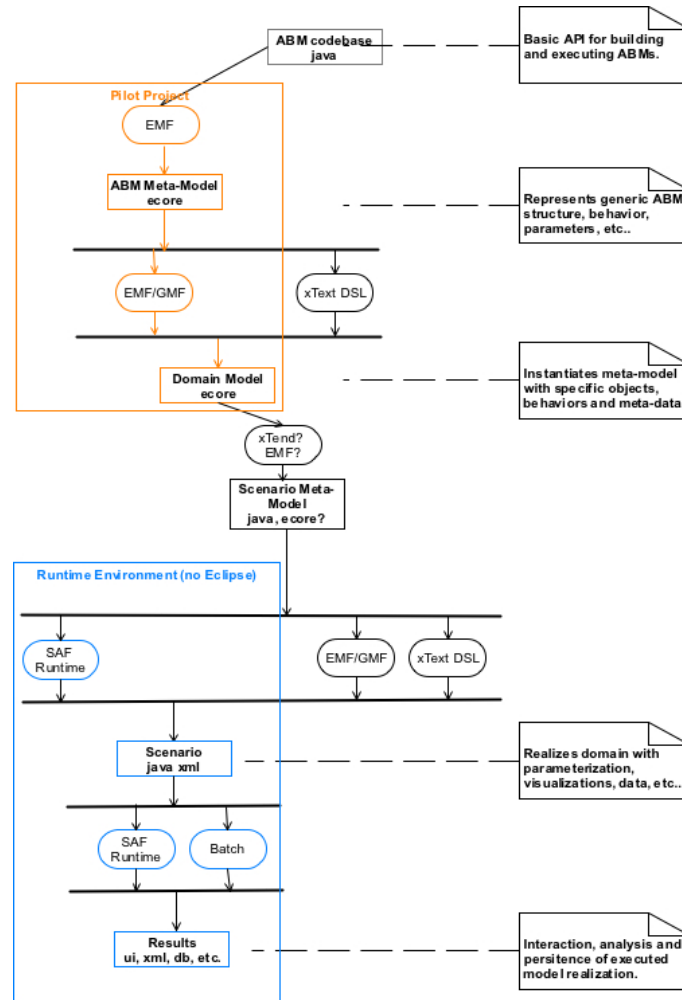
---

5 The question of whether we are and should be developing first and foremost a "model of" or a "language for" agent-based modeling is deep and complex. We will not address these issues here, but interested readers could start with Fowler's companion posting on "Language Workbenches and Model Driven Architecture."

6 It is important to note that Repast S and its related tools are still under development. This paper presents the most current information at the time it was written. However, changes may occur before the planned final release.

7 Credit or blame to Howe, "Ensemble" (Parker) has also been proposed in keeping with an analogizing context while "milieu" and "gestalt" remain candidates.

Here we take special note of the upper half of the diagram. The core modeling component is targeted for our pilot. We will describe the initial release and the runtime component more fully in a later paper. Briefly, we plan on using Xtend, a model transformation language, to

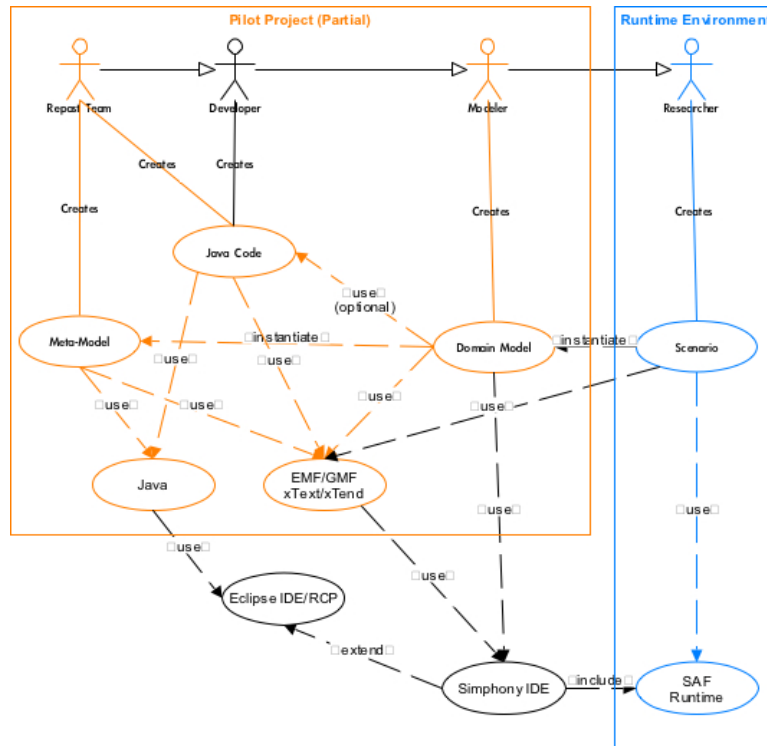


**FIGURE 1** Score activity diagram

infer a scenario model from the domain model. For example, suppose that our modeler includes a 2-dimensional grid in his or her model. The domain model will then include a slot in the scenario meta-model<sup>8</sup> for defining for each scenario what size the scenario's grid should be or the allowed range of sizes for the scenario's grids. Next the modeler or researcher can define specific scenarios to run interactively (e.g., using real-time visualization) or in batch mode (e.g., a parameter sweep). The results of these models can then be externalized and the models used to guide interpretation of these results.

<sup>8</sup> It is certainly possible that we will find that a straightforward object model could suffice for the definition of arbitrary scenarios, but the services provided by EMF (e.g., built-in editors) and persistence alone seem to justify its use here.

Figure 2 provides greater detail about how different kinds of users will interact with the Repast S system as a whole and the dependencies between different tools. The exposition that follows focuses mainly on the block of the system that represents the modeling process itself. First, the Repast team or outside contributors develop the core Repast S API, the Score meta-model which makes use of the Repast S API, and the templates that will be used for model transformation.



**FIGURE 2** Score use-case diagram

There is an intermediate role for a developer who develops models and Repast S extensions, continuing to use the Java API directly. The modeler is a new entity who, up to now, has been imagined mostly as a renaissance person who is both a master of his or her domain and a software developer. The approach discussed in this paper allows modelers to focus on their domain rather than on software development by letting them create meta-models directly by using the graphical and textual tools described above. Finally, we have the researcher. With the existing toolsets, this scholar must find a graduate student somewhere to “code up” his or her model. Guided by the meta-information provided by the Score and Scenario meta-models, the Repast S system can now provide much more sophisticated tools to aid in model exploration and offers the researcher a natural path to the role of modeler.

Fundamentally, the Score meta-model captures the specification needed to generate a complete agent model. The meta-model is depicted in a UML object diagram in Figure 3. The key to understanding such meta-models is to recognize that they are not describing agent simulations or even the object structure for agent simulations, but a model sufficient to inform the creation of such simulations. For example, SField does not represent an actual agent field, but the meta-data we would need to describe some agent variable (e.g., the name of the field, its data

type, its allowed ranges, and so on). Consider further that these values are themselves represented in Ecore, this meta-model's meta-model, as EAttributes carrying similar meta-data. This field then would be used along with other meta-data to create an object model for the actual agent with an actual “happiness” member variable. Similar descriptions can be given for other model meta-objects. In some cases, however, there is a direct one-to-one correspondence. For example, for every SValueLayer defined in the model, there will be an actual ValueLayer in our domain agent model.

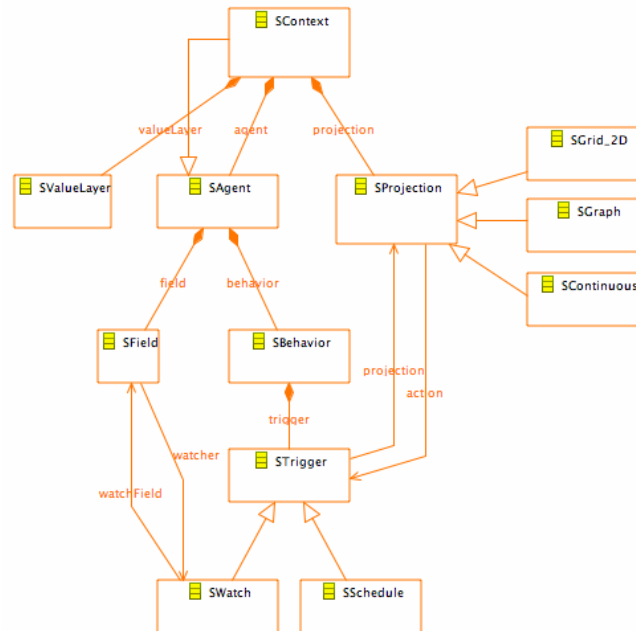


FIGURE 3 Score meta-model

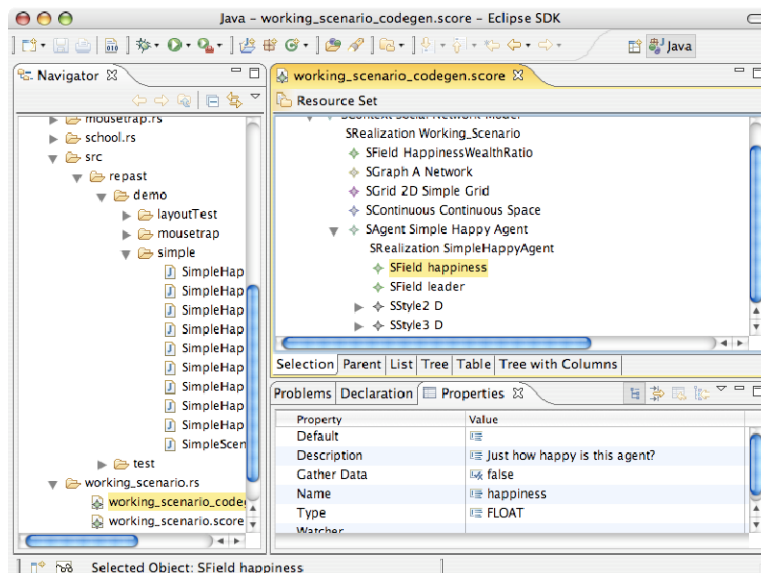


FIGURE 4 Simple Score tree editor

## “SCORE” IN ACTION

We provide a very preliminary walk-through for the initial incarnation of Score, excluding from this discussion many of the tools envisioned above, such as a DSL textual language. First, an agent modeler defines a simple model using either a basic tree-based modeler or an XML editor. In fact, it is instructive to compare the two.

In the tree editor, the modeler can add various components to a hierarchy (see Figure 4). As discussed elsewhere, the Repast S context approach is naturally hierarchical, but more complex graph relationships certainly can exist beneath the hierarchy (North et al. 2006b). Future GUI modeling tools may provide more complex editors for these structures, but for reasons outlined in Howe et al. (2006), we believe that hierarchies defined through contexts will remain the most natural and effective representation. Ultimately, the user’s model is stored in an XML/XMI file, and users can also edit this file directly and employ a referenced schema to enforce basic constraints, which assure that the XML file can be parsed into an instantiation of the meta-model.

After we have developed a complete model, we can use EMF’s built-in XMI persistence scheme to easily create a Java object instantiation. This step is so much simpler than typical approaches to XML-Java and mappings that it is essentially transparent. At this point, we have objects for all of the meta-data (e.g., types, agent names, built-in components, etc.) that we need for creating a complete model specification as defined in the initial Repast S roadmap.<sup>9</sup> That, coupled with the flexibility we will gain in evolving the model specification into the future, may alone be worth the initial effort in integrating EMF into our tool-chain. However, we should note that we have only implemented a prototype of the template code that will drive our code-generation; we will leave detailed discussion and examples for a future paper.

By using the oAW workflow engine, we should be able to integrate the model development process into a seamless development experience within the Eclipse environment. The Check language and Recipe framework provide a way to give model developers real-time feedback, immediately notifying them of incorrect or ill-advised edits and providing meaningful hints on how to correct them.

The meta-model will next be run through our Xpand2 templates, producing Java source code for our actual agents. For example, from the above SimpleHappy model we could create source code for a SimpleHappyAgent, including lines for variable and getters and setters for the agent based on text patterns we have defined. But we could do much more than that; our agent could have direct references to its parent context(s) and their state and could very efficiently support listeners (watchers) for agent state, and we can generate parent context-level variables (e.g., minHappiness, maxHappiness, happinessDistribution) to randomly initialize each SimpleHappyAgent’s state. We note, however, that our agents can be plain old Java objects and thus be transparent and full-fledged members of the Java world; they can support JavaDoc; contain inspection, reflection, and other persistence mechanisms; and integrate into production

---

<sup>9</sup> There are two modes that can be used in realizing the actual code used in a running model. The more sophisticated usage we are focusing on in this paper involves generating the code for model components and then loading them into the runtime environment. But we can also simply use “pre-built” Java classes directly by setting the realization mode attribute to “LOAD.” This usage is the one supported by the walk-through to this point.

and enterprise environments and toolsets. Finally, we can use the model to generate efficient and appropriate implementations for runtime tools such as visualizers and data collection, etc.<sup>10</sup>

Most importantly, we can realize arbitrarily complex behaviors for our agents that can be transformed into native Java syntax. In our first implementation, we plan to provide a simple but quite focused Java-like syntax for specifying many of these behaviors. We anticipate that sophisticated models for rich and expressive behavior will be a very active area of future research and development.

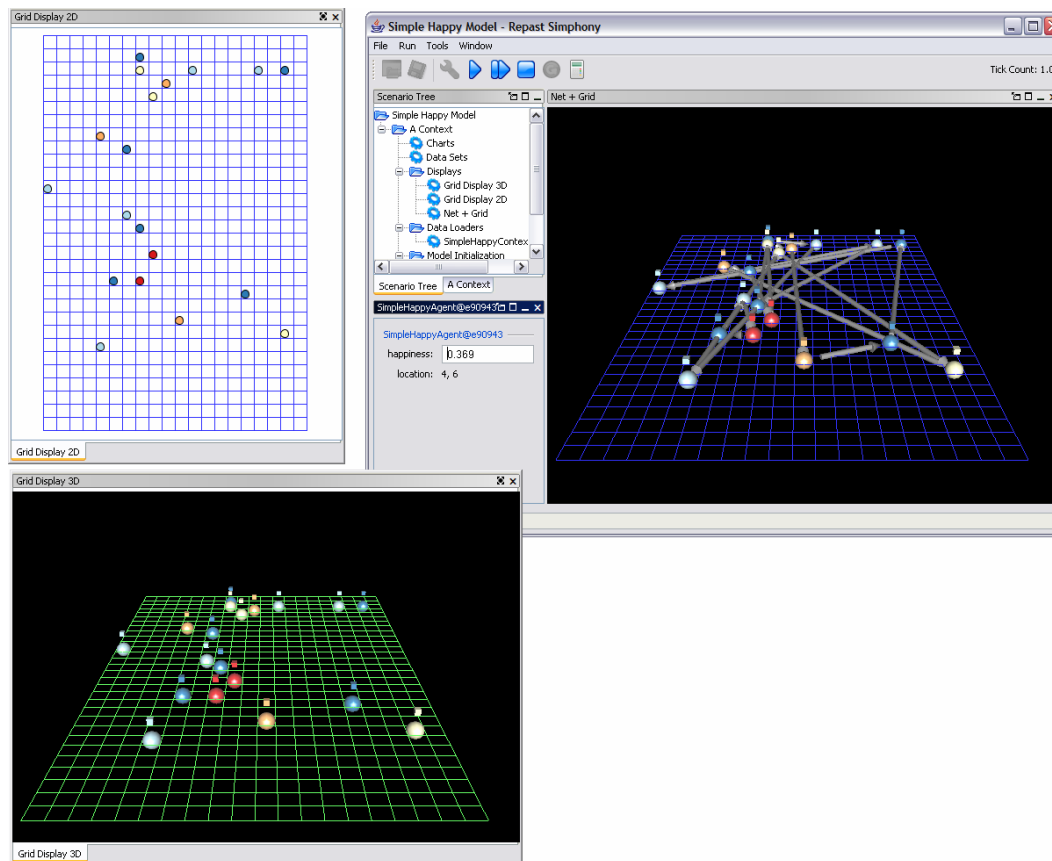
At this stage, the meta-model will also be transformed through Xtend into a scenario Ecore meta-model that can then be used to provide the same kinds of tools as outlined above but at the level of model instantiation and execution. Researchers will be able to define realizations within their own models (e.g., by adding specific agents to a particular context, specifying batch constraints, and so forth). As mentioned above, some of this work can be (and has been) done using a generic object model, but employing a meta-level specification for each model may provide much greater leverage and cleaner abstractions for developing such supporting tools. For example, we could eliminate Java reflection and easily be able to generate XML Schemas for individual model batch runs.

```
<?xml version="1.0" encoding="UTF-8"?>
<repast.score:SContext xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:repast.score="http://repast.score" name="Social Network Model">
  <realization package="repast.demo.simple" className="Working_Scenario"
path="..\plugins\repast.test.models" mode="GENERATE"/>
  <field name="HappinessWealthRatio" type="FLOAT" default=".45"/>
  <projection xsi:type="repast.score:SGraph" name="A Network"/>
  <projection xsi:type="repast.score:SGrid_2D" name="Simple Grid" xSize="10"
ySize="10"/>
  <projection xsi:type="repast.score:SContinuous" name="Continuous Space"/>
  <agent name="Simple Happy Agent">
    <realization className="SimpleHappyAgent" path="" mode="GENERATE"/>
    <field name="happiness" description="Just how happy is this agent?"
type="FLOAT" default=""/>
    <field name="leader" default=""/>
    <style xsi:type="repast.score:SStyle2D">
      <realization className="SimpleHappyNodeStyle2D" path="" mode="LOAD"/>
    </style>
    <style xsi:type="repast.score:SStyle3D">
      <realization className="SimpleHappyNodeStyle" path="" mode="LOAD"/>
    </style>
  </agent>
</repast.score:SContext>
```

**FIGURE 5** An example Score XML file

<sup>10</sup> Beyond the structural and behavioral aspects described above, there are many more intriguing code-generation dimensions that are far outside the scope of this paper.

Finally, the researcher will actually execute the model by using either a stand-alone runtime as shown in Figure 6, a batch runner, or a even a possible model runtime environment fully integrated with the Eclipse environment. Data and other artifacts collected during the model run can then be interpreted and manipulated using scenario and model meta-data.



**FIGURE 6** Running model

## CONCLUSION

We advocate an approach to software development that may strike some as arcane and overly complex. The possibilities for “meta-confusion” may seem endless, and the specter of proliferating “little languages” may cause panic for those who believe that ubiquitous standards and generic UML are the answer to every problem. However, the approach discussed in this paper is not an “all-or-nothing” proposition. Developers interested in Java-level programming can continue to use the Repast Java APIs directly, mixing in generated code as appropriate and making use of a powerful new agent-specific language as time and interest permit. And modelers without traditional programming skills will finally have a complete set of tools for generating agent-based models and sharing the insights these models provide into our complex world.

## ACKNOWLEDGMENT

This work is supported by the U.S. Department of Energy, Office of Science, under contract W-31-109-Eng-38.

## REFERENCES

- Gulyás, L., T. Kozsik, and J.B. Corliss, 1999, “The Multi-Agent Modeling Language and the Model Design Interface,” *The Journal of Artificial Societies and Social Simulation*, Vol. 2, No. 3.
- Howe, T.R., N.T. Collier, M.J. North, M.T. Parker, and J.R. Vos, 2006, “Containing Agents: Contexts, Projections, and Agents,” *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*, co-sponsored by Argonne National Laboratory and The University of Chicago, Argonne, Sept. 21–23 (in press).
- Inchiosa, M.E. and M.T. Parker, 2002, “Overcoming Design and Development Challenges in Agent-based Modeling Using Ascape,” *Proceedings of the National Academy of Sciences*, **99**: Suppl. 3, 7304-7308.
- North, M.J., T.R. Howe, N.T. Collier, and J.R. Vos, 2005, “The Repast Symphony Runtime System,” *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, ANL/DIS-06-1, co-sponsored by Argonne National Laboratory and The University of Chicago, Oct. 13–15 (in press).
- North, M.J., N.T. Collier, and J.R. Vos, 2006a, “Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit,” *ACM Transactions on Modeling and Computer Simulation*, vol. 16, issue 1, pp. 1–25.
- North, M.J., T.R. Howe, N.T. Collier, J.R. Vos, and M.T. Parker, 2006b, “Spaces, Places, and the Lernaean Hydra of Agent-Based Modeling,” *North American Association for Computational Social and Organizational Science 2006 Conference*, Argonne National Laboratory, Argonne, IL.
- Parker, M., “Ascape: Abstracting Complexity,” *Brookings Institution Report*, March 2000, and *SwarmFest Proceedings 2000*, Utah State University, Logan Utah; available at [http://www.brookings.edu/es/dynamics/models/ascape/20000301\\_ascape.htm](http://www.brookings.edu/es/dynamics/models/ascape/20000301_ascape.htm).
- ROAD (Repast Organization for Architecture and Design), 2006, Repast Home Page, Chicago, IL.; available at <http://repast.sourceforge.net/>.
- Voelter, M., B. Kolb, S. Efftinge, and A. Haase, 2006, “From Front End to Code—MDS in Practice,” available at <http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSInPractice/article.html>.
- Wilensky, U., 1998, *NetLogo*, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

