Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

NANOVERSE: A CONSTRAINTS-BASED DECLARATIVE FRAMEWORK FOR RAPID AGENT-BASED MODELING

David Bruce Borenstein

Lewis-Sigler Institute for Integrative Genomics
Princeton University
Carl Icahn Laboratory
Princeton, NJ 08544, USA

ABSTRACT

Agent-based models (ABMs) are ubiquitous in research and industry. Currently, simulating ABMs involves at least some imperative (step-by-step) computer instructions. An alternative approach is declarative programming, in which a set of requirements is described at a high level of abstraction. Here I present the a fully declarative methodology for the automated construction of simulations for ABMs. In this framework, called "Nanoverse," logic for ABM simulations is encapsulated into predefined components. The user specifies a set of requirements describing the desired functionality. Additionally, each component has a set of consistency requirements. The framework iteratively seeks a simulation design that satisfies both user and system requirements. This approach allows the user to omit most details from the simulation specification, simplifying simulation design.

1 INTRODUCTION

1.1 The need for descriptive modeling

Agent-based models (ABMs) constitute one of the most widely used categories of simulation technology. ABMs represent a system as an ensemble of autonomous actors (or "agents"). These agents interact with one another according to predefined behaviors. The set of defined behaviors may be unique to each individual agent, or common to a class of agents. ABMs are widely used for academic research in the fields of ecology, epidemiology and social science (Eubank et al. 2004, Grimm et al. 2005, Gilbert 2008). Commercial and governmental uses include business analytics, supply chain management, and civil and military planning (Fox, Barbuceanu, and Teigen 2000, Cioppa et al. 2004, Delre et al. 2007, Zheng, Zhong, and Liu 2009).

ABMs provide a link between local and global dynamics. The modeler defines local interactions. As these interactions play out, global patterns often become evident. ABMs can be used to predict large-scale patterns based on smaller-scale processes. Conversely, by selecting rules that recapitulate observed large-scale processes, modelers can make predictions about the underlying local interactions. In either case, effective use of ABMs requires deep insight into the process at hand—a body of knowledge wholly disjoint from the computer expertise required to actually build models. This challenge is exacerbated by the introduction of spatial structure, where topological details can have major implications for emergent behavior (Durrett and Levin 1994, Borenstein et al. 2013).

The predominant representation of ABMs outside of software is a standardized rubric of model features, called the 'Overview, Design concepts, and Details" (ODD) approach (Grimm et al. 2006). In an ODD specification, agent-based models are described in terms of their structure and temporal dynamics. Notably, computer logic is minimized in the ODD specification: the rubric focuses on what the model does, rather than how a programmer chose to accomplish it. Simultaneously, there have been efforts to develop a

logic-based framework for the general description of simulations (Guizzardi and Wagner 2010). Most recently, software design patterns–standardized architectural units with commonly recognized names–have been generalized to the domain of agent-based models for the purpose of standardizing ABM vocabulary (Beck and Cunningham 1987, North and Macal 2011).

Here, I present the methodology behind a declarative ABM framework, Nanoverse, that is based on the principle of description. Nanoverse is an open-source domain-specific language developed by this author, with accessory components contributed by multiple individuals (see Acknowledgments). A companion paper (under review) will present a case study for Nanoverse concerned with the microbial Type VI Secretion system, a cell-to-cell weapon found in gram-negative bacteria. Additional details are also available at the framework's website (http://nanover.se/). This article is based on Chapter 4 of the author's PhD thesis, and draws heavily on that text (Borenstein 2015).

In existing approaches, model design is closely linked to simulation design. The model deals with the properties of the agents and the world they occupy. When do they act? What information can they incorporate into their choices? The simulation, on the other hand, is a computer program capable of actualizing the model and integrating it over time (Miller and Page 2007). For the most part, existing tools are simulation frameworks: it is up to the user to first envision a model and then articulate a process for simulating it.

The key innovation behind Nanoverse is that it structures ABM implementation as a configuration problem. Rather than specify step-by-step rules, the user imposes constraints (requirements) on the model. The platform then uses this specification to find a configuration of predefined components that satisfies these constraints. By replacing step-by-step (imperative) computer instructions with a (declarative) description of a model's properties, simulation design can be brought closer in line with the ways in which ABMs are discussed.

1.2 Imperative approaches to ABM design

Simplifying ABM development has been the focus of much research and development. Much of this research has focused on general-purpose software tools for spatially structured ABMs. Most ABM software tools have introduced expressive computer languages (or language extensions) created for the specific purpose of ABM simulation (Railsback, Lytinen, and Jackson 2006). By far the most successful project has been NetLogo, which extends the educational programming language LOGO (Feurzeig, Papert, and Lawler 1969) to a large library of agent-specific structures and actions. NetLogo (Wilensky 2004) has been widely adopted in academic research, and remains popular after nearly two decades of continuous use. An extreme form of this approach is purely visual programming, as in StarLogo TNG (Klopfer and Begel 2007) and Scratch (Maloney et al. 2010). These K-12 educational tools make modeling easier by representing imperative statements as visual blocks.

Paradoxically, the simplicity of these LOGO-derived tools means that complex models can be challenging to express. Another tool, GAMA (Grignard et al. 2013) seeks to address some of these limitations by providing straightforward facilities for GIS and multi-level models. GAMA utilizes a fluent, object-oriented language called GAML. GAML automates many aspects of model design, but still requires the user to specify and manipulate data structures. Frabjous, another ABM-specific language, overlays temporal tracking onto the functional programming language Haskell to create a flow-based programming paradigm analogous to that of commercial package Anylogic, discussed below (Schneider et al. 2012, Vendrov et al. 2014).

Other imperative tools for complex simulations include Java libraries such as MASON (Luke 2005) and Repast (North et al. 2013). These tools each provide a powerful, object-oriented framework within which to build and simulate ABMs atop a discrete-event scheduler. However, these tools require proficiency with general-purpose programming languages such as Java (Gosling et al. 2013). Previously mentioned AnyLogic uses a variety of UML-like charts to represent the states and actions of agents, which it then translates to Java code (Borshchev, Karpov, and Kharitonov 2002). While highly accessible, this approach

is essentially analogous to imperative programming since it requires the user to define a sequence of logical actions and reactions.

1.3 A component-based architecture for ABMs

Many agent-based models can be simulated using a common set of strategies. NetLogo, GAMA, Repast and StarLogo all provide an extensive library of common logical pieces; often the only programming task is to unite these pieces in a manner appropriate to the model. Nanoverse extends this concept further, by hierarchically building up components from a pool of subcomponents. By repeatedly applying this idea, it is possible to define agent-based models from a relatively small body of simple units. Since all imperative logic would be encapsulated in these units, the user's task becomes one of describing conceptual relationships, rather than computational tasks. This is the principle behind component-based (or "modular") software engineering (Bachmann et al. 2000, Reinhold 2014).

There is precedent for a component-based approach to simulation: SimKit provides a structure for building and distributing reusable imperative blocks, which can then be composed programatically or visually (Buss 2002, Buss and Sanchez 2002, Buss and Blais 2007).To the author's knowledge, there has been no effort to leverage the declarative nature of component-based software in order to present simulation design as a configuration task. This approach opens up a wealth of existing strategies for simulation implementation, as configuration problems are a cornerstone of knowledge engineering (Wielinga and Schreiber 1997).

Configuration problems can be solved using constraint satisfaction approaches. In a constraint satisfaction problem, a "solution" is any value which satisfies every specified constraint. The goal of a constraint-based configuration scheme is to satisfy both the requirements imposed by the user and the requirements imposed by the selected sub-components, given a set of available options.

A straightforward approach to constraint satisfaction is backtracking. In a backtracking algorithm, solutions are tested sequentially against the first constraint, being globally eliminated if they violate it. Once a solution is found, the algorithm recurs on the next constraint. If no solution satisfies a constraint, it "backtracks" to the previous constraint, which resumes its search (Wielinga and Schreiber 1997). By specifying a sequence of default subcomponents, the backtracking strategy is sufficient to configure a single component of a simulation, such as a spatial structure. An entire simulation can be specified by nesting constraint satisfaction problems together, in a strategy known as composite constraint satisfaction (CCS) (Sabin and Freuder 1996). This approach has previously been used to automate other software configuration tasks, such as the deployment of complex software systems (White et al. 2007). Constraint programming is closely related to logic programming, for which backtracking is extensively employed. As such, logic programming languages such as Prolog include features related to these approaches (Russell and Norvig 2009). Nanoverse draws on these techniques to create a declarative simulation environment layered atop the Java virtual machine (Lindholm et al. 2013).

This paper describes part of an ongoing effort to create a constraint-driven, spatially explicit agent-based modeling framework. This framework, called Nanoverse, is being prototyped in stages. The first stage, a working mock-up of which is available online (http://nanover.se), is a component-based simulation environment that is functionally similar to GAMA or MASON. This paper concerns the second stage of the prototype, currently under development, consisting of a multi-stage compiler. The paper begins with a brief synopsis of the runtime environment into which the compiler instantiates simulations. The second part describes the architecture of the Nanoverse compilation pipeline.

2 RUNTIME SCHEME

The Nanoverse runtime consists of a network of loosely coupled components. The primary subsystems of the runtime are a collection of topologies called "layers" and a discrete event scheduler (Fishman 2001).

The layer encapsulates all topological information, and the schedule encapsulates all scheduling information. Mutation of the simulation state is accomplished through scheduling events with a relative waiting time, which is subsequently resolved by the schedule. Likewise, specific changes to the environment are specified relative to a particular agent. As such, agents remain completely agnostic to the global state of the simulation.

In order to accomplish this, events have callbacks that request specific changes to their locale. Agents have a rule table mapping specific conditions to the triggering of certain events. When a simulation event runs, it notifies the layer, which notifies all affected agents to consult their rule table. The simulation ends when the event queue is depleted or another terminal condition is met. The loose coupling of simulation components allows for the use of a constraints-based compiler system, which in turn allows us to move away from imperative programming.

3 COMPILE SCHEME

3.1 Overview

The Nanoverse compiler prototype consists of a four-stage compilation pipeline, ultimately leading to a discrete-event runtime for spatially explicit agent-based models (Fig. 1). The first stage of the pipeline is a parser that interprets a hierarchical source code into an abstract syntax tree (AST). This abstract syntax tree has no semantic information about the structure of an agent-based model; it reflects only the grammar of the user's specifications. The second stage uses a hierarchy of symbol tables to convert the abstract symbol tree into a semantically rich hierarchy of "build nodes." These build nodes roughly correspond to the Java objects that will represent the simulation in memory. The third stage of the pipeline is the backtracking constraint solver, which is used to interpolate unspecified properties of the simulation. This is done by treating the user's specifications as additional constraints on an ordered sequence of defaults, with over- or underdefined specifications leading to an error. Finally, the completed build tree is visited breadth-first in order to instantiate all nodes into Java objects. The top-level object then triggers the execution of the simulation.

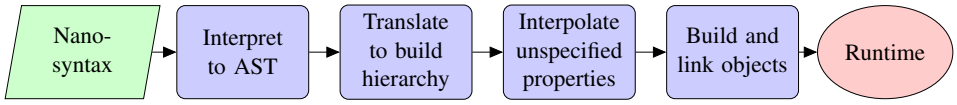

Figure 1: Nanoverse compilation pipeline

3.2 The Nanosyntax environment

The user writes Nanoverse model descriptions using a hierarchical grammar called "Nanosyntax." Nanosyntax was influenced by the JSON object specification, which is used to serialize data for transmission between internet servers and clients (Crockford 2006).

Nanosyntax consists of three types of nodes: "primitives," "references" and "assignments." Primitives are basic data types, such as numbers and strings. References specify an identifier or a block of identifiers. Assigmments map an identifier to a reference. Additional structures are allowed for mathematical operations, but these are converted internally to the other node types. These three elements are sufficient to specify an arbitrary hierarchy of members in a concise and readily intelligible way.

Nanoverse project specifications consist of a nested ensemble of constraints, or explicit requirements concerning the properties of the simulation. Any requirements left unspecified are subsequently interpolated from a set of defaults to match the constraints that were specified.

Nanosyntax is fully declarative: the only purpose of the source code is to describe what should be done, rather than how it should be implemented. The properties to be specified correspond directly to encapsulated operational units, or components, of the simulation's business logic. As a result of interpolation, Nanosyntax

is also minimal: the source code contains only the requirements of interest to the user. The user can therefore begin running simulations with very little code. By iteratively overriding defaults, the user can then build up the behavior of the simulation until it reflects all desired functionality.

As an example, consider the "StupidModel" reference model developed by (Railsback, Lytinen, and Jackson 2006). The first of 16 instances of the model consists of a population of 100 agents that diffuse around a 100x100 rectangular lattice. The anticipated Nanosyntax for an even simpler model, which consists of a single agent diffusing around a 32x32 rectangular lattice, is as follows:

```
initially:
    scatter:
        description:
            Agent:
                do: Behavior {
                    action: wander;
                    every: 1.0;
                    until: time >= 100.0;
                };
```

The Nanosyntax representation of the model describes the entire system in a single statement block. Time is specified in arbitrary units. Absent a specific geometry requirement, the system defaults to a 32x32 rectangular lattice with absorbing boundary conditions. Since the only action – diffusion, or `wander`ing—is encoded in the definition of the agent itself, there is no need to define a main loop. The `wander` operation itself does have subcomponents dealing with destination selection and collision resolution, but these are also handled with interpolated defaults. Specifying an alternative boundary condition, lattice geometry or arena shape would take one additional line apiece.

Existing frameworks require far more code to accomplish the same goal. MASON and Repast both require the user to define diffusion from first principles, instantiate a 2D arena, and place the agents using a random number generator; moreover, all of this must be done in Java. NetLogo eliminates the need for low-level programming, but still requires explicit instructions for each operation involved (Railsback, Lytinen, and Jackson 2006). GAMA requires that the user first define a geometry and a neighborhood structure, then the conditions for an ongoing behavior (or "reflex") representing movement. The user then defines a visual representation of the agent, and a display mode for the visual representation (Amouroux 2014). The GAMA approach is similar to that of Nanoverse, except that Nanoverse is designed to resolve many of the specified details that are required in GAMA.

The Nanoverse compiler parses Nanosyntax using the parser generator ANTLR4 (Parr 2014). ANTLR4 generates a parse tree. The Nanoverse compiler then translates the Nanosyntax parse tree into an abstract syntax tree (AST), an example of which is shown in Fig. 2. Nanoverse employs a heterogeneous AST: different nodes are used for each of the three basic data types (Parr 2010). By distinguishing between data types, the AST provides structural information that simplifies the next process in the pipeline: semantic analysis.

3.3 Adding semantic information

After parsing user syntax, Nanoverse constructs a partial representation of model semantics (Fig. 3). This partial semantic model, known as the "object node hierarchy" or "build hierarchy," encodes all requirements explicitly specified by the user. Nanoverse constructs this hierarchy through the use of a graph of symbol tables.

At their most basic, symbol tables are mapping functions from a text symbol to some other value (Grune et al. 2000). A compiler for an imperative language will typically create a single symbol table at each level

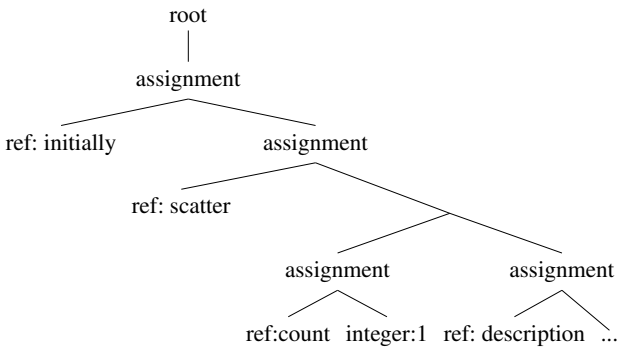

Figure 2: Top portion of an abstract syntax tree.

of contextual scope to associate identifiers with values (Cooper and Torczon 2011). Nanoverse symbols, on the other hand, represent system components: i.e., loosely coupled subsystems that supply specific functionality for the containing system, and which themselves depend on further subsystems (Bachmann et al. 2000). As such, Nanoverse uses a symbol table for every component and component class.

To encode its rule base, the Nanoverse compiler constructs two classes of symbol tables: *resolving* symbol tables (RSTs) and *instantiable* symbol tables (ISTs). RSTs narrow a particular identifier to a specific subclass of an expected class. ISTs resolve the names of specific subsystems required to instantiate an object of a specific class. Object translation proceeds by alternating between these two symbol table classes.

After translation, the user's requirements have been translated into a hierarchy of constraints. The compiler must now determine whether and how a simulation can be instantiated from the user's specifications. The user's constraints may be expressly incompatible, or they may imply further requirements that are incompatible. In these cases, the model is overdetermined. On the other hand, the user may have omitted required fields (i.e., fields with no default values). If this happens, the model is underdetermined. Assuming neither an overdetermined nor underdetermined model, the compiler's next task is to interpolate sufficient constraints to fully determine the model's configuration.

3.4 Interpolation and construction

Nanoverse organizes a simulation into a hierarchy of components. "Components" in Nanoverse are equivalent to "primitives" in NetLogo or GAMA (Wilensky 2004, Grignard et al. 2013), except that most components are not "primitive" in the sense of being discrete, atomic wholes. Rather, a Nanoverse component may have an arbitrary number of subcomponents, which may likewise have subcomponents of their own. Components are only loosely coupled to their subcomponents—often by a single method—facilitating interchange. Interchangeable components are at the heart of the configuration-based approach.

The configuration of Nanoverse components is accomplished through the hierarchical solution of local constraints. Each subcomponent has its own constraints. These constraints determine whether the component is compatible with the existing partial configuration, and which subcomponents can be supplied to it. Additionally, the subcomponents for a given component may depend on one another, and are thus supplied as additional constraints on the subcomponent. Associated with each subcomponent is one or more defaults, which are given in order of preference. Each default may imply its own set of constraints. If the user has specified a particular value for a subcomponent, the specified subcomponent (and its implied constraints) replaces the default list. Component configurations are then solved depth first until a total solution has been found, or it is determined that no solution exists (Fig. 4).

In a constraint satisfaction problem, solutions are often obtained through a backtracking scheme. A backtracking scheme consists of a recursive algorithm. Let $v_0, ...v_i$ represent the values that must be specified, and let D_i represent the domain of solutions for v_i. In addition, there exists a set C of constraints

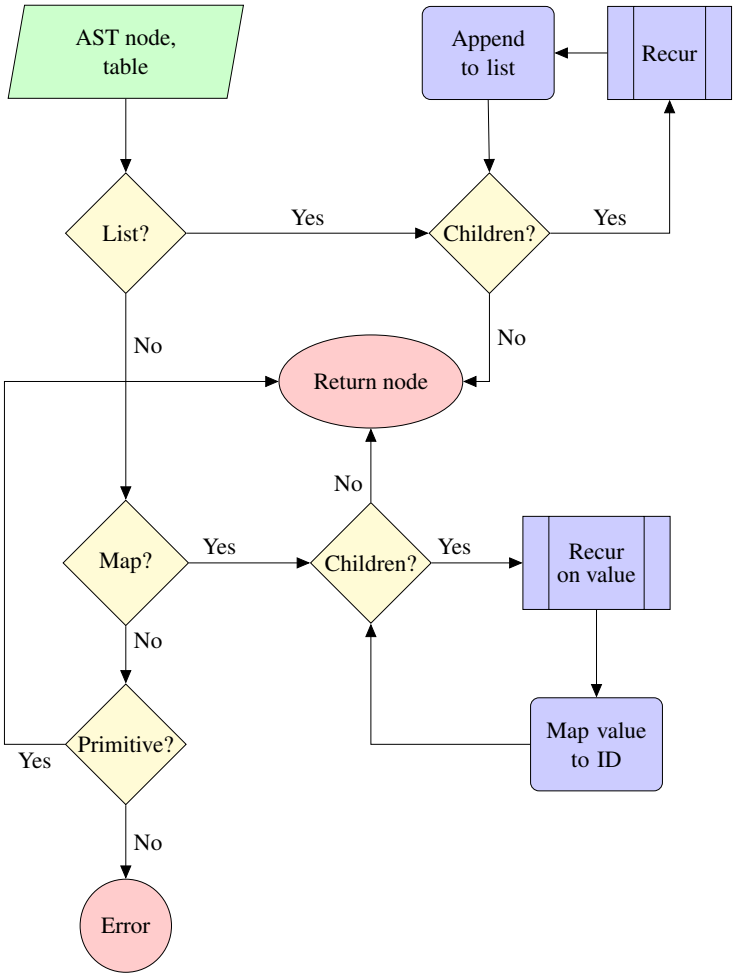

Figure 3: A flowchart representing the translation of an abstract syntax tree node to an object node.

on the solution set. The backtracker begins by seeking a value $v_0 = d_0 | d_0 \in D_0$ that satisfies the relation $C \cap v_0 = d_0$. If such a value is found, the algorithm recurs on v_1, D_1. For the nth recursion, the algorithm seeks a value of $v_n = d_n | d_n \in D_n$ that satisfies $C \cap (v_k = d_k \forall k \leq n)$. If this relation cannot be not satisfied, the algorithm returns failure (Russell and Norvig 2009).

In Nanoverse, the constraints represent the specific requirements of particular subcomponents. The constraint set C is therefore not globally constant. However, for any given component, the only constraint is that all of its subcomponents are legal, given their dependencies. Thus, a subcomponent can verify constraint satisfaction by verifying that all of its subcomponents can find legal instance values. For simple subcomponents with only one possible default value, such a check is relatively straightfoward. More complex subcomponents must perform their own interpolation step. This component-dependent interpolation step is encapsulated in the "Valid?" decision node in Fig. 4.

Instantiation proceeds like interpolation. Each component has its own instantiation method, which builds any helper objects as necessary. For the most part, these helper objects are themselves components, albeit not user-specified ones. That is, they are only loosely coupled to the parent component, and they are automatically configured based on the properties of the parent component. Helper components include getters and setters from other runtime objects, which serve the same role as public method calls in traditional APIs.

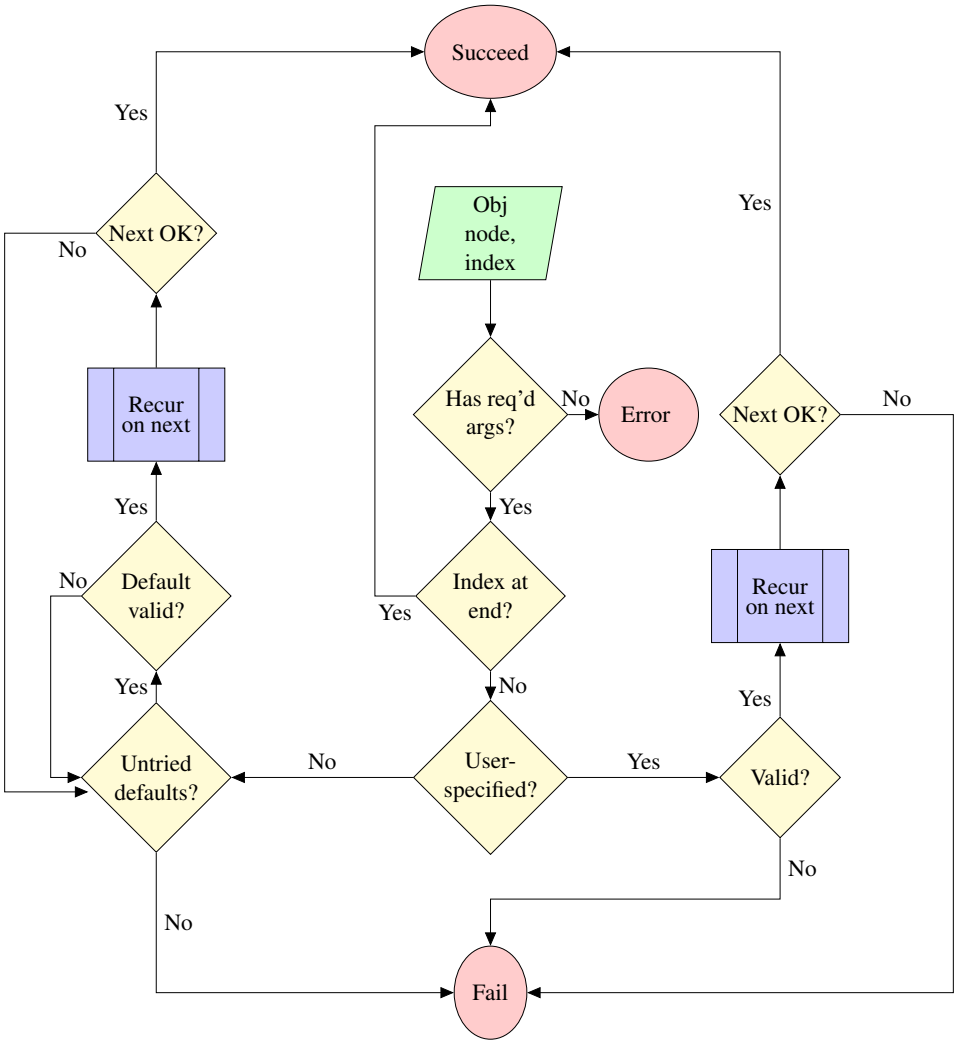

Figure 4: Flowchart representing the constraint satisfaction process used to interpolate unspecified user parameters.

4 CHALLENGES AND FUTURE DIRECTIONS

4.1 Engineering considerations

Component-based software design has been widely incorporated into many software platforms, most recently in the form of Java's "Project Jigsaw" (Bachmann et al. 2000, Reinhold 2014). Difficulty of maintenance is a challenge that is common to all of these approaches (Weyuker 1998). Component-based software essentially shifts some of the user's responsibilities onto the developer. Rather than define logic, the user builds pre-fabricated logical components into the desired configuration. The power and utility of a component-based platform is therefore limited by the breadth and quality of these pre-fabricated components. The developer must provide both the runtime logic and any steps required for the component to compile.

In the case of Nanoverse, these steps include specifying acceptable sub-components and the order of preference for those components. This implies a larger codebase than analogous, imperative systems. Nanoverse is also fully declarative, unlike hybrid declarative-imperative languages like GAMA. As a fully declarative language, the only possible business logic is that which is defined in an existing component.

Many agent-based models share some similar ideas, especially in Nanoverse's primary domain of spatially explicit ABMs. Special cases abound, however, and new questions lead constantly to new model designs. How can Nanoverse accommodate advanced use cases without complicating the simple ones? Perhaps the most straightforward solution is to incorporate an imperative sub-language, which would be backed by a simple interpreter. This would have the added benefit of allowing model changes on the fly, and even self-modifying code, which can be used for evolutionary simulations.

Testing is another major challenge. It is often desirable to test a component both in isolation and in its intended context. However, the number of possible contexts for any given component is limitless: as user models (and the component library) grow, the same component can be nested deep in a hierarchy of other parts. When the assumptions of components are in contradiction, unexpected behavior can result. These risks can be managed through the judicious use of consistency checks, strong interface contracts, and exhaustive unit testing (Crnkovic and Larsson 2002). More practically, cross-validation of benchmark simulations against other simulation platforms can help to assure that Nanoverse's behavior is consistent with expectations.

4.2 Default generality

In the Nanoverse prototype, the only planned constraints have to do with logical compatibility. For example, a spatially explicit model taking place in a hexagonal arena cannot employ a periodic boundary condition, because two of the six sides would remain unmatched. Likewise, a rectangular lattice cannot employ a hexagonal arena. This lowers the skill threshold required for use, but it does not handle another important class of constraint: preventing the selection of many parameters that, while technically not in conflict, may produce unexpected behavior.

There are many situations in which the user would expect different defaults based on his or her selections, even if one default could technically satisfy all cases. This is particularly true for spatially structured systems. Consider, for example, the resolution of collisions. What should happen if an agent that is scheduled to move has no vacant space into which it can go? Different application areas will require different answers. A good solution in a forest fire model (e.g., intensify the fire) is different from that of a microbial model (push the existing occupant away). In a traffic simulation, meanwhile, opposite direction movement could lead to many different interpretations and outcomes (Ljubović 2009).

The user must specify how to choose a destination, and, if collisions are possible, how to resolve them. If the user specifies that destinations must include occupied locations, there must be a rule for resolving a collision. That said, permitting occupied spaces is compatible with a resolution strategy of "throw an error on collisions," though this is unlikely to be desired. One approach to domain-specific defaults is the ability to specify custom "default sets," and to inherit these elements as domain-specific libraries.

5 CONCLUSION

The Nanoverse compiler has the potential to simplify the process of building agent-based models. This greater ease can benefit both novice and experienced users: the user need not specify any parameters whose defaults are satisfactory. With the introduction of component libraries and default sets, Nanoverse can also function as a medium for the transmission of expert knowledge concerning model design: domain expertise can be encoded into defaults and component behaviors, which can then be used by novice modelers. As with many agent-based modeling platforms, the same approach can be used to simplify the design of interactive systems, such as games.

The strict hierarchical structure of the Nanoverse language provides several benefits. The Nanoverse compiler already exploits the most important of these: the availability of algorithms to interpolate missing nodes. Hierarchies are also easy to visualize, e.g. using a zooming user interface (Bederson and Meyer 1998). The strict separation of concerns required for hierarchical design also simplifies compiler design, which facilitates optimization of program flow. Finally, a component-based design results in a highly

decoupled library of component symbol tables. These symbol tables can be used to generate documentation for the Nanosyntax language automatically and as the language evolves.

The nanoverse compiler is under active development. The first goal is to port all runtime functionality from the interpreted Nanoverse prototype, including modular topology and continuum-valued fields, to the compiler-based edition. Following that, I plan to provide an automatic documentation system and publish it to the Nanoverse website. Beyond that, I will focus on addressing the limitations of the language by introducing user-defined variables, user-defined constraints and defaults, object orientation, and code importation.

ACKNOWLEDGMENTS

The author gratefully acknowledges Ned S. Wingreen for his support and guidance throughout the development of the Nanoverse framework, as well as Annie Maslan and Daniel Greenidge for their contributions to the code base.

REFERENCES

Amouroux, Edouard 2014. "GAMA tutorial: StupidModel". https://code.google.com/a/eclipselabs.org/p/gama/wiki/StupidTutorialModel1v14. Accessed 30 March 2014.

Bachmann, F., L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. 2000. *Technical Concepts of Component-Based Software Engineering, vol. 2*. Bedford, MA: SEI Joint Program Office.

Beck, K., and W. Cunningham. 1987. "Using Pattern Languages for Object-Oriented Programs". Submitted to the *OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming*.

Bederson, B., and J. Meyer. 1998. "Implementing a zooming User Interface: experience building Pad++". *Software: Practice and Experience* 28:1101–1135.

Borenstein, D. B., Y. Meir, J. W. Shaevitz, and N. S. Wingreen. 2013, May. "Non-Local Interaction via Diffusible Resource Prevents Coexistence of Cooperators and Cheaters in a Lattice Model". *PLoS ONE* 8 (5): e63304.

Borenstein, D. 2015. "A multi-agent approach to the evolution of microbial populations in the presence of spatially structured social interaction". (Doctoral dissertation, in press). Princeton, NJ: Princeton University.

Borshchev, A., Y. Karpov, and V. Kharitonov. 2002. "Distributed simulation of hybrid systems with AnyLogic and HLA". *Future Generation Computer Systems* 18:829–839.

Buss, A. 2002. "Simkit: component based simulation modeling with Simkit". *Proceedings of the Winter Simulation Conference*, 243-249.

Buss, A., and C. Blais. 2007. "Composability and component-based discrete event simulation". *Proceedings of the Winter Simulation Conference*, 694–702.

Buss, A., and P. Sanchez. 2002. "Building complex models with LEGOs (Listener Event Graph Objects)". *Proceedings of the Winter Simulation Conference*, 1:732–737.

Cioppa, T. M., T. W. Lucas, and S. M. Sanchez. 2004. "Military applications of agent-based simulations.". *Proceedings of the Winter Simulation Conference*.

Cooper, K., and L. Torczon. 2011. *Engineering a compiler*. 2nd ed. Morgan Kauffmann.

Crnkovic, I., and M. Larsson. 2002. *Building Reliable Component-Based Software Systems*. London: Artech House.

Crockford, D. 2006. The application/json media type for javascript object notation (JSON). https://tools.ietf.org/html/rfc4627 Accessed 18 June 2015.

Delre, S. A., W. Jager, T. H. A. Bijmolt, and M. A. Janssen. 2007. "Targeting and timing promotional activities: An agent-based model for the takeoff of new products". *Journal of Business Research* 60:826–835.

Durrett, R., and S. A. Levin. 1994. "The importance of being discrete (and spatial)". *Theoretical Population Biology* 46 (3): 363–394.

Eubank, S., H. Guclu, V. S. A. Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang. 2004. "Modelling disease outbreaks in realistic urban social networks.". *Nature* 429 (May): 180–184.

Feurzeig, W., S. Papert, and B. Lawler. 1969. "Programming-languages as a conceptual framework for teaching mathematics. Final Report on the First Fifteen Months of the LOGO Project". Cambridge, MA: Bolt Beranek and Newman.

Fishman, G. 2001. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. New York, NY: Springer.

Fox, M. S., M. Barbuceanu, and R. Teigen. 2000. "Agent-Oriented Supply-Chain Management". *International Journal of Flexible manufacturing systems*. 12.2-3 (2000):165–188.

Gilbert, N. 2008. *Agent Based Models*. Thousand Oaks, CA: Sage.

Gosling, J., B. Joy, G. Steele, G. Bracha, and A. Buckley. 2013. "The Java Language Specification: Java SE 8 Edition". Redwood City, CA: Oracle America.

Grignard, A., P. Taillandier, B. Gaudou, D. A. Vo, N. Q. Huynh, and A. Drogoul. 2013. "GAMA 1.6: Advancing the art of complex agent-based modeling and simulation". *Lecture Notes in Computer Science*, Volume 8291 LNAI, 117–131.

Grimm, V., U. Berger, F. Bastiansen, S. Eliassen, V. Ginot, J. Giske, J. Goss-Custard, T. Grand, S. K. Heinz, G. Huse, A. Huth, J. U. Jepsen, C. Jørgensen, W. M. Mooij, B. Müller, G. Pe'er, C. Piou, S. F. Railsback, A. M. Robbins, M. M. Robbins, E. Rossmanith, N. Rüger, E. Strand, S. Souissi, R. a. Stillman, R. Vabø, U. Visser, and D. L. DeAngelis. 2006. "A standard protocol for describing individual-based and agent-based models". *Ecological Modelling* 198:115–126.

Grimm, V., E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, H.-H. Thulke, J. Weiner, T. Wiegand, and D. L. DeAngelis. 2005, November. "Pattern-oriented modeling of agent-based complex systems: lessons from ecology". *Science* 310 (5750): 987–91.

Grune, D., H. E. Bal, C. J. Jacobs, and K. G. Langendoen. 2000. *Modern Compiler Design*. Wiley.

Guizzardi, G., and G. Wagner. 2010. "Towards an ontological foundation of discrete event simulation". *Proceedings of the Winter Simulation Conference*, 652–664.

Klopfer, E., and A. Begel. 2007. "StarLogo TNG: An Introduction to Game Development". 1–15.

Lindholm, T., F. Yellin, G. Bracha, and A. Buckley. 2013. "The Java Virtual Machine Specification: Java SE 8 Edition". Redwood City, CA: Oracle America.

Luke, S. 2005. "MASON: A Multiagent Simulation Environment". *Simulation* 81:517–527.

Ljubović, V. 2009. "Traffic simulation using agent-based models". *Information, Communication and Automation Technologies*.

Maloney, J., M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. 2010. "The Scratch Programming Language and Environment". *ACM Transactions on Computing Education (TOCE)* 10 (4): 1–15.

Miller, J. H., and S. E. Page. 2007. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton, NJ: Princeton University Press.

North, M. J., N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. 2013. "Complex adaptive systems modeling with Repast Simphony". *Complex Adaptive Systems Modeling* 1 (1): 3.

North, M. J., and C. M. Macal. 2011. "Product design patterns for agent-based modeling". *Proceedings of the Winter Simulation Conference*, 3087-98.

Parr, T. 2010. *Language implementation patterns*. 1st ed. Pragmatic Bookshelf.

Parr, T. 2014. *The Definitive ANTLR4 Reference*. 2nd ed. Pragmatic Bookshelf.

Railsback, S. F., S. L. Lytinen, and S. K. Jackson. 2006. "Agent-based Simulation Platforms: Review and Development Recommendations". *Simulation* 82 (9): 609–623.

Railsback, S. F., S. L. Lytinen, and S. K. Jackson. 2006. "Supporting Files for: Agent-based Simulation Platforms: Review and Development Recommendations". http://condor.depaul.edu/slytinen/abm/. Accessed 28 March 2015.

Reinhold, M. 2014. "JSR 376: JavaTM Platform Module System". Technical report, Oracle Corporation. https://www.jcp.org/en/jsr/detail?id=376. Accessed 26 March 2015.

Russell, S., and P. Norvig. 2009. *Artificial Intelligence: A Modern Approach*. 3rd ed. Upper Saddle River, NJ: Prentice Hall.

Sabin, D., and E. C. Freuder. 1996. "Configuration as Composite Constraint Satisfaction". *Proceedings of the AI and Manufacturing Research Planning Workshop*, 153–161.

Schneider, O., C. Dutchyn, and N. Osgood. 2012. "Towards frabjous: a two-level system for functional reactive agent-based epidemic simulation.". *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, 785–790.

Vendrov, V., C. Dutchyn, and N. Osgood. 2014. *Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling.. Social Computing, Behavioral-Cultural Modeling and Prediction*, 385-392. New York, NY: Springer.

Weyuker, E. J. 1998. "Testing component-based software: A cautionary tale". *IEEE Software* 15 (October 1998): 54–59.

Wielinga, B., and G. Schreiber. 1997. "Configuration-design problem solving". *IEEE Expert-Intelligent Systems and their Applications* 12:49–56.

Wilensky, U. 2004. "NetLogo: A simple environment for modeling complexity". *International conference on complex systems*. 16–21.

White, J., D. C. Schmidt, K. Czarnecki, C. Wienands, G. Lenz, E. Wuchner, and L. Fiege. 2007. "Automated model-based configuration of enterprise Java applications". *Proceedings of the IEEE International Enterprise Distributed Object Computing Workshop, EDOC* (1): 301–312.

Zheng, X., T. Zhong, and M. Liu. 2009. "Modeling crowd evacuation of a building based on seven methodological approaches". *Building and Environment* 44:437–445.

AUTHOR BIOGRAPHY

DAVID BRUCE BORENSTEIN, Ph.D., is a postdoctoral research associate at the Lewis-Sigler Institute for Integrative Genomics at Princeton University. He recently completed his doctorate in Quantitative and Computational Biology, also at Princeton. Before coming to Princeton, David worked in genome sequencing at the Broad Institute of MIT and Harvard. David began developing the Nanoverse simulation framework to facilitate the study of population evolution in microbes. His previous papers have focused on the evolutionary impact of interbacterial warfare and cooperation. In addition to his scientific training, David holds a degree in philosophy. dbborens@princeton.edu.