NISAC Agent-Based Laboratory for Economics (N-ABLE™): Overview of Agent and Simulation Architectures

Eric D. Eidson and Mark A. Ehlen

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000

Approved for public release; further dissemination unlimited.
NISAC Agent-Based Laboratory for Economics (N-ABLE™): Overview of Agent and Simulation Architectures

Eric D. Eidson and Mark A. Ehlen
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-1138

Abstract

The NISAC Agent Based Laboratory for Economics (N-ABLE™) is an agent-based micro-simulation tool that models complex interdependencies between economic and infrastructure sectors. The agent and simulation architectures have been designed so that large data-driven simulations can be developed quickly and conducted on Sandia's massively parallel computing clusters. This report documents the agent architecture for running large-scale simulations, the economic agent classes that allow for creating microeconomic entities of varying resolution, and the simulation environment that allows for automated large-scale simulation.
This page is intentionally blank
1 Introduction

1.1 Background

The National Infrastructure Simulation and Analysis Center (NISAC) analyzes the impacts of infrastructure disruptions and interdependencies on regional and national economic security. Funded through the Department of Homeland Security, NISAC is developing and applying a wide range of tools that model urban, regional, and national infrastructure dynamics.

One of these tools, the NISAC Agent-Based Laboratory for Economics (N-ABLE), is an agent-based microsimulation tool that models complex interdependencies between economic and infrastructure sectors. N-ABLE agents are relatively small pieces of computer code that model economic agents such as:

- firms using material inputs, labor, capital, electric power, telecommunications, transportation, and banking to produce and sell products;
- households shopping for and purchasing products;
- electric power companies producing and selling power;
- buyers and sellers in electric power spot markets;
- telecommunications firms that process phone calls and data transmissions;
- financial traders in stock and commodity exchanges;
- transportation firms selling and providing shipping services;
- manufacturing firms that manage production and distribution supply chains; and
- banks processing checks for customers.

N-ABLE, and agent models in general, have proven to be very useful in designing, implementing, and analyzing models of complex economic systems, in which small actions by a limited number of individuals (firms, households, government agencies) cause economy-wide, “emergent” behavior. Each agent individually acts out simple rules; groups of these agents interact randomly with each other. The results can be very complex economic market, economic production, and infrastructure dynamics. In this sense, agent-based models are a natural fit for modelling economies where the simple act of a consumer purchasing a good is, on its own, insignificant, but the collective demand pull of many consumers purchasing many goods drives an entire economy in complex and non-intuitive ways.

Agent-based economic models can characterize complex market dynamics that are otherwise difficult to analyze mathematically, say, with traditional closed-form calculus solutions. For example, N-ABLE makes Adam Smith’s “invisible hand” quite visible: N-ABLE consumers purchase goods by shopping among vendors of the good, looking for low prices. When a consumer is quoted a price, he determines whether he could save more by continuing to shop (at some personal cost). Firms are then driven to compete on prices, driving efficiencies in the market and economy as a
By modelling agents in their simple consumer role – making decisions on what to buy and when – N-ABLE simulations can model complex real-world market dynamics.

N-ABLE is influenced greatly by the ten years of Sandia work in agent-based economic models. In 1993, Aspen was first developed to analyze economic growth, financial markets, and monetary policy (Sprigg and Ehlen[13], Pryor et al[11], Basu et al[4], Basu and Pryor[3]). Through later research, Aspen was completely recast in C++, dividing the core agent functionality into (1) AgentLib classes, which provide fundamental services such as agent creation, messaging, and event processing; and (2) Aspen classes, which build economic firms and markets from the AgentLib classes. This Aspen model was then applied to electric power markets (Barton et al[1]). More recently, the N-ABLE economic classes, built from scratch from the AgentLib classes, add infrastructure such as water or electric power utility systems, transportation, and telecommunications network (Barton et al[2]) capabilities. To date, N-ABLE has been applied successfully to analytical studies of electric power demand-management policies, large industrial supply chains, hazardous chemical transport networks, and regional economies. Looking forward, N-ABLE can now model well the economic impacts of disruptions to and interdependencies between infrastructures that are primary to the U.S. economy.

As illustrated by Figure 1, N-ABLE is actually a suite of programs that collectively provide the ability to model and simulate, including building a specifications file that represents a model of an economy and infrastructures that support it (through the client user interface), starting a simulation or a set of, say, Monte Carlo simulations (through the SimRunner), broadcasting real-time data to multiple viewers and users (through the SimStreamer), and storing output data in a portable XML format. Data gathering, simulation creation, and external simulation control operate from an Windows user interface; simulation, data archiving and real-time data streaming occur in a Unix environment. Communications between platforms use SOAP and socket protocols.

---

1 Adam Smith noted specifically of sellers, and economic agents in general, “by directing that industry in such a manner as its produce may be of the greatest value, he intends only his own gain, and is in this, as in many other cases, lead by an invisible hand to promote an end which was no part of his intention.” (Smith[12])

2 At that time the AgentLib classes were called “AspenLib.” The name has been changed to remove any spurious correlation between this N-ABLE work and other ongoing, separate Aspen work.

3 See http://n-able.sandia.gov for current status of these and other studies.

4 This Unix requirement (e.g., Linux, FreeBSD) is due to some highly customized and tuned code that handles the heaviest processing by the SimStreamer and SimRunner; future extensions to real-time network models will, for performance reasons, likely involve Linux-based pipes or other speed tools. The internal N-ABLE simulation engine, however, can still be compiled in, say, Windows, simulations can be started at the command line using manual input file submission, and simulation results can be viewed in the “*.SNAPS” output data file. That is, the SimRunner and SimStreamer are not required to run N-ABLE; they serve to provide real-time viewing and collaboration, both
operating the simulations primarily in Linux, N-ABLE can leverage well one of the benefits of the underlying AgentLib classes: the ability to run both on local single-processor (PC) or remote massively parallel machines, depending on the size of the simulation.

1.2 Purpose and Scope

The purpose of this report is to give a detailed overview of the N-ABLE model’s object structure, mechanics, and supporting functionality, so that those running N-ABLE simulations or working with the N-ABLE code itself better understand how the model works. The report describes in detail the AgentLib and N-ABLE object classes used to construct N-ABLE simulations. The report focuses primarily on the model portion of the N-ABLE suite, specifically the computer code architectures used to model the agents. Throughout the report, we use the term “N-ABLE” to refer to the modelling program itself and not to the applications that prepare and submit the inputs (client user interface, SimRunner) and serve up and visualize the outputs (SimStreamer, client use interface).

Section 2 describes the computing issues that motivate the particular architecture of the AgentLib classes. Section 3 describes the AgentLib library, including fundamental base-class agent types, how agents are generated, how they are given “turns” to do their tasks, and how they organize their data for “snapshotting” to output files. (The AgentLib library is designed to be stand-alone, that is, to generate agent models other than N-ABLE.) Section 4 then describes how the AgentLib classes are used to make N-ABLE models and simulations. Section 5 outlines some examples of N-ABLE agents, and Section 6 some example simulations. Finally, Section 7 displays the N-ABLE v1.1 client user interface and how it is used to help users make, run, and review the output of simulations.

Throughout the document, we use the strict convention that a capitalized object, such as “Seller,” denotes a class within either AgentLib or N-ABLE, while an uncapitalized object (“seller”) denotes the object in abstract.

2 Motivations for Agent Framework

The AgentLib set of classes, or AgentLib framework, was developed to support rapid development of agent-based models that can run on a variety of hardware platforms – from desktop PCs to the massively parallel multiprocessor machines available at Sandia National Laboratories. To make this possible, the framework must offer a number of basic capabilities, including: populating an agent simulation, running and controlling the simulation, and gathering the output data. The framework must also provide advanced capabilities, such as the distributing of agents in one simulation across processors in parallel computing environments and coordinating their interactions. Moreover, this framework must work correctly on and consistently across a range of hardware and software platforms.

The AgentLib classes meet these criteria through the following properties and capabilities:

- they are written in a portable, widely used C++ programming language;\(^5\)
- they can generate arbitrary models of agents and other objects, using data input files;

of which are strong benefits in the analytical environment of DHS.

\(^5\)AgentLib is based on a portable subset of standard C+++, ISO/IEC 14882:1998(E).
Figure 2: AgentLib Classes

- they serialize objects (such as agents) and move them between processors;
- they coordinate parallel processors so that simulation results are deterministic, i.e., the results don’t depend on unpredictable and uncontrollable factors within the hardware and operating system; and
- they generate and store results in a portable XML format.

By gathering all the mundane details of generic agent functionality in the AgentLib framework, N-ABLE programmers can concentrate on implementing within N-ABLE highly specific economic functionality.

3 AgentLib Classes

The AgentLib framework consists of the set of classes shown in Figure 2. The Model class, which has overall simulation control, uses the ModelGen and ObjGen classes to instantiate a simulation’s model and agents. Simulations use agents created from the Agent abstract class and identified through their AgentID or ObjectID information. The agents schedule tasks and messages to each other using the Event, Task, and Message classes, and are distributed across multiple computer processors using the Transfer classes. Output data is created with the Snapshot class. Each of the primary and supporting AgentLib classes is described in turn.

3.1 Model Class

The Model class is the primary AgentLib class for initiating a simulation, maintaining control during simulation, and exiting. During a simulation, the Model acts as the central “go-to” place where, particularly in multiprocessor systems, agents and other objects can access global, simulation-level functions and data, such as the Model Calendar and global Random Number Generator.

Agent actions are scheduled on the Model’s Calendar as events. As the Model moves its simulation clock forward it reads which events are to occur at that particular point in time and then

Specifically, simulations that use the same random number seed, same number of processors, and same input file will generate the same output results.
invokes them. To create and schedule actions, each AgentLib agent, like a real person, plans his day down to the individual action, and then as a way of remembering to carry out the task, puts the tasks on the Model Calendar, in the form of Events that it will perform. As a day progresses, each agent is reminded as to the next thing it must do and then it does it. Sometimes in the process of doing one thing, it realizes something else must be done, or is told by another agent to do something, and thus schedules it on the Calendar as well.

The Model class ensures that the sequence of events in simulations are repeatable: simulations based on the same input data, same number of processors, and same random number seed will produce the same results. This is particularly important in simulations where small, local perturbations can cause system-wide changes in economic dynamics. For example, consider a simulation in which two agents approach the same car dealership in the same time period, each with events ready to purchase what turns out to be only one remaining car. The result – which agent actually gets the car – depends on which event the Model executes first. If two events from two agents are scheduled to occur at the same time, the order in which they occur is determined by the random number seed, the interactions between processors, the priority queue, and other factors. If the overall simulation results are to be deterministic and repeatable, the Model must always cause the events to occur in the same order. In contrast, if the results could be different every time, important types of sensitivity analysis could not be performed.

3.2 Model Generation (ModelGen) and Object Generation (ObjGen)

To create agents for a simulation, the Model class uses the Model Generation (ModelGen) class to read the input data file that lists the number and types of agents to be created. To do this, the ModelGen class invokes from Object Generation (ObjGen) portions within each of the agent classes an object generator function, which first specifies the type of data needed to generate an instance of this agent and, second, actually uses the specified data to generate an instance of the agent. From the perspective of an AgentLib agent class itself, say AgentX, AgentX’s ObjGen prepares the ModelGen so that it can know to possibly encounter an AgentX data type in the inputs file and if so, then know how to create an AgentX-based agent.

Chronologically, at runtime the Model activates the ModelGen to read the input file. Before actually reading the input file, the ModelGen first uses the TypeReg classes (described below) to find the object types associated with and required by all ObjGens. As the input file is read, its component tags tell the ModelGen the quantity and characteristics of each agent; the ModelGen then looks up the ObjGen associated with the agent and instantiates one or more of these agents in memory. The ModelGen then fills each agent ObjGen with the specifications data found in the file. Once the entire specifications file is read, the ModelGen uses the ObjGens to generate objects (e.g., agents) which are passed back to the Model, which then executes the simulation.

3.3 Agent, AgentID, and ObjectID Classes

All agents in an AgentLib-based simulation are based on the AgentLib Agent class, an abstract class (i.e., it cannot be used directly) that has the necessary properties and functions to identify itself and others (with AgentIDs) schedule tasks, and send messages.

---

7While AgentLib treats actual time as arbitrary, N-ABLE uses the minute unit of time as the smallest time resolution. For example, N-ABLE electric power infrastructure providers and users produce and consume power on a minute-by-minute basis.
An N-ABLE or other AgentLib-based simulation must be able to uniquely identify each of its agents and objects, for example, so that it can deliver a message from Agent 1 to Agent 2. In general, in single-processor systems a C++ pointer is sufficient to uniquely identify an object, since all objects reside in the same address space – that is, all the memory that can be “addressed” by the processor – and no two objects occupy the same addressable slot in memory. However, a program running on parallel or multi-processor hardware has access to potentially N address spaces for N processors. Since a C++ pointer can only refer to one address space, an agent residing in one processor cannot “see” agents in other processors.

The AgentLib uses two classes, AgentID and ObjectID, to uniquely identify all objects in a simulation, regardless of the address space in which they reside. AgentIDs are given to all agents (and only to agents), while ObjectIDs are given to any other object that needs a unique identifier in all address spaces.\(^8\)

### 3.4 Event, Task, and Message Classes

AgentLib Events are actions taken by N-ABLE agents and other objects to schedule an action on the Model Calendar. The Event class provides the necessary functionality for creating and managing all simulation events, including prioritizing, scheduling, and invocation.\(^9\)

Every event listed on the Model Calendar is an instance of some class derived directly or indirectly from the Event class. AgentLib provides two primary sub-classes of Event: Task and Message. AgentLib agents start their activities with a Task Event (e.g., “review stock of supplies”); in the course of performing the Task, the agent sends a Message (e.g., “order new supplies”, or "schedule future review stock of supplies"). These Messages are processed by the Calendar and their responses sent, whereby these responses are processed, and so on. The list of types of all Event class events and their relative priorities are shown in Table 1.

As illustrated in Figure 3, The Model Calendar processes events by first accepting them from agents and objects. At the beginning of each time period, the Calendar then sorts the events according to the relative priorities specified in Table 1. The Calendar then processes each event in order.\(^10\)

---

\(^8\) In particular cases, however, a collection of AgentLib objects do reside in the same address space or processor. For example, the fundamental EconomicAgent (discussed in the N-ABLE Classes section below) keeps all of its internal objects (e.g., buyers, sellers, production, accountant) in the same address space; C++ pointers are therefore sufficient for the EconomicAgent to access them.

\(^9\) This AgentLib Event class preserves some of the terminology and behavioral model of the original Aspen event code, but invokes an event-driven model more contemporary with object-oriented programming. Compared with Aspen, the AgentLib Message Event is a Message-delivery Event – “sending a Message” is scheduling its delivery on the Model calendar, which then delivers the message at the scheduled time. Aspen did not use an event-driven framework for scheduling events; instead, the time line was divided into large time steps (i.e., each day was one time step), and each time step was divided into stages. The first stage was the task stage in which all agents performed their daily tasks; the agents sent messages to one another. The next stage was the first message stage; agents would receive messages sent during the task stage and could send responses. Next, the second message stage was performed, and so on until all agents had processed all messages and had sent no new responses. Further, on parallel hardware, the Model class uses an implicit staging system, which, like Aspen, buffers all Messages sent during one “round.”

\(^10\) Like Aspen, AgentLib’s Event class yields deterministic results on parallel hardware with relatively low communications overhead per time step, but is more flexible, particularly with agents that work on different time scales (e.g., some N-ABLE agents do something every minute, while others do something once a month).
Table 1: Types of Calendar Events, In Descending Order of Priorities

<table>
<thead>
<tr>
<th>Calendar Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaturalLanguage</td>
</tr>
<tr>
<td>TaskEvent1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>TaskEventN</td>
</tr>
<tr>
<td>MessageEvent1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>MessageEventN</td>
</tr>
</tbody>
</table>
3.5 Bulletin Board Class (BBoard)

AgentLib provides a bulletin board class, BBoard, so that agents have structured means of finding each other (such as a buyer finding a seller in a particular market). The BBoard class is essentially a listing or directory of the AgentIDs of simulation agents; it allows agents to be categorized or stratified by economic market, social class, or physical location, thereby providing a wide range of contexts for agent interactions.

Centralized information, such as a bulletin board, that is part of an AgentLib-based multi-processor simulation could be maintained one of two ways: either in the Model object itself, or as a distributed agent. To remain highly flexible in the number and type of bulletin boards that can be instantiated in AgentLib-based simulations, the BBoard class derives from the Agent class, which, as an agent, can be transferred (via the Transfer classes described below) across processor nodes, and receive and send Messages.

As an example of its use, the N-ABLE Market class inherits from BBoard class to provide markets of buyers and sellers who need each others’ services. At simulation start, each buyer and seller sends a message to the Market/BBoard/Agent object to add it to the directory of agents. As Sellers modify their prices, they send Messages to the BBoard object to update their listed price. Buyer agents looking for a Seller of the product will request the listing of Sellers and their prices and then contact a Seller using its listed AgentID. As another example, N-ABLE also uses the BBoard-based SocialNetwork class and a Socializer agent to create social, i.e., non-market, networks of agents.

3.6 Snapshot Class

Agent simulations generally produce a lot of output data — either there are few agents with a lot of detail, or tens of thousands of agents with medium detail. Data production and archival must then be highly structured and flexible. AgentLib provides a Snapshot class designed to export agent and other object data in a flexible and portable XML document format. N-ABLE simulations, as an example, stream this data to a local file and to a separate SimStreamer application which serves this data to any and all N-ABLE client user interfaces. While the Snapshot class is internally relatively complex, use of the Snapshot system as easy as supplying the required registration in the N-ABLE programming code.

As illustrated in Figure 4, the Model Calendar makes periodic calls to agents for them to “pose” for snapshot-data output; each agent also can immediately tell internal subagents and objects to pose as well (only those data members that the agent has registered for posing are included in the Snapshot).

Because some of these objects are complex sub-objects in their own right (such as list of purchase records), each sub-object has its own Snapshot registration. In a beneficial sense, such registration of complex types for Snapshots can be arbitrarily deep. At the bottom of a registration chain is the registration of some numerical type (e.g., the selling price of a good) suitable for display on a graph.

This design is possible (and efficient) because the Snapshot registration procedure builds a collection of pointers-to-members. Pointers-to-members are odd C++ constructs that point to a member’s location in a struct or class, which is very different from a traditional pointer, which points to the memory address of a specific member of a specific object. By learning and remembering member locations, the Snapshot system can simply take an object, determine what class it was
instantiated from (usually statically), apply the pointer-to-member and get a traditional pointer to
the data item. That item is then written to the stream in the proper format.

This design, however, comes at several costs. The pointers-to-members are *contra-variant* in
the sense that a pointer-to-member can not point to a member in a derived class (compared with
*co-variance* or *polymorphism*, where a pointer to a base class can also point to a derived class),
AgentLib has a complex set of template wrappers designed to capture type information and store
the pointers. This template system forces the compiler to generate a lot of small classes, which
increases compile time and adds a small amount of “code bloat.”

Overall, though, the smaller registration functions and centralized file format design trades
off very well against delegated writes mainly because the file format itself is very complex. By
centralizing the design, we avoid frequent debugging efforts looking for a problem that is distributed
among an ever-changing list of classes.

3.7 Supporting AgentLib Classes

3.7.1 Type Registration (TypeReg)

Every object (including every agent) in an N-ABLE model is an instance of some pre-defined type.
For example, the number 3.1415926 may be defined as a float or double, while the string “Hello
World” may be defined as a character array. So that type names declared in N-ABLE object
classes are portable, i.e., the types mean the same thing regardless of the particular compiler,
the type registration system (TypeReg) includes additional type information to many types found
in the program. In particular, the TypeReg system assigns each type a visible name that can
be used to portably identify the type in input files (by ModelGen), when transferring objects
between processes (by the Transfer system), and when sending simulation data to output files (by
the Snapshot system). The type registration system also identifies class hierarchy of *polymorphic
objects* (such as agents) and determine when and where such objects can be created and become
associated with other objects. This is especially useful when generating new models.

Although much of the code that defines the TypeReg system is found in AgentLib, the actual
registrations are distributed in the various N-ABLE files that implement the types being registered.
By localizing such registrations, we avoid a giant registration function that quickly becomes out
of date. Indeed, we can customize the N-ABLE program for some problems simply by adding or
removing the relevant files that define agents and the type registration system will automatically
know what is (and by extension, what is not) available.
Massively parallel computing environments, while proving additional computing power to agent simulations, require additional computer code to handle both the spreading of agents across these nodes at the beginning of simulation and potentially during a simulation (dynamic load balancing). To move agent objects and other simulation objects across nodes, they must be serialized and de-serialized in an orderly manner.

Normally, objects like Agents exist in computer memory in a form that is machine-dependent – that is, the sizes, layouts and locations of data that make up the object are tied to a particular machine. In order to transfer an object from one machine to another, the object first has to be serialized into a intermediate representation that can safely be moved between machines. Then on the receiving end, the object has to be deserialized or expanded back into a usable object – one that is machine dependent on the receiving machine. Also, separate programs running on the same machine (i.e., "processes") each have their own view of memory (i.e., an address space) – so while the data sizes and layouts may be the same for two copies of the same program on the same machine, the differing address spaces require objects to be serialized for exchange.

The Transfer system is a collection of classes that provide a way to serialize and deserialize objects. It provides support for serializing objects into a variety of intermediate representations (such as a binary format, a text format and an XML format). We usually choose the binary format because it is fastest. Occasionally the binary format causes problems when exchanging between machines that are "too different,” so we’ll use the Text format instead. If someone has to inspect the serialized data, we’ll prefer to use the XML format because it’s easier to read (but it is also several orders of magnitude slower than the binary format).

Our main use of the transfer system is to move objects between parallel processes. On a parallel computer, each CPU or processor runs a separate copy of the parallel program (i.e., separate processes). Each process has its own address space, and hence requires serialization to exchange objects. In our case, we usually want to exchange Message objects (which are like letters that Agents send one another). In such cases, a process will use the transfer system classes to serialize the object into a character string and send that string to the destination process. The destination process will receive the string then use the transfer system classes to deserialize the object into its own memory space. Once the process is complete, an object is said to have been "transferred” between the two processes.

The name "Transfer system” is a slight misnomer – the Transfer system performs the serialization and deserialization steps, but relies on something else for the actual transfer of the serialized representation. For example, to transfer between MPI processes, we would serialize into a string, then use a pair of MPI Send and Receive calls to transfer the string between the processes.

The Transfer System consists of a base class called Transfer and many overloaded functions named transfer. Every built-in type (i.e., char, int, float, double, etc.) has a transfer function, a set of "primitives” that every other transfer function builds upon. The primitive transfer functions call a corresponding virtual function in the Transfer base class. Classes derived from the Transfer class (for example the paired input/output versions of BinStorage, TextStorage, and XMLStorage) override the virtual functions of the base class to store the corresponding built-in type in the given format. (For example, BinStorage reads or writes the data in binary format, while TextStorage stores the data in text format.) The transfer function for a type is used both for incoming and outgoing transfers – the actual behavior (i.e., inputting or outputting) is determined by the Transfer
3.7.3 Random Number Generator

So that large multi-processor simulations are deterministic (repeatable), AgentLib provides a centralized Random Number Generator class that abstracts the process of obtaining pseudo-random numbers. Currently, the Random Number Generator generates its pseudo-random numbers through a lagged Fibonacci sequence, using an algorithm recommended by Knuth\[9\] as a general-purpose generator (other algorithms could be used in the future). One advantage of this algorithm is that each possible seed yields effectively independent sequences with very long periods; in contrast, traditional algorithms typically yield sequences with relatively short periods and where any seed merely positions the generator somewhere in the sequence.

3.7.4 Classifier

N-ABLE sellers currently use a genetic algorithm (GA) to search out market prices that give their company the greatest profits.\(^\text{11}\) This GA uses AgentLib Classifier system, which includes a set of sensor functions that “measure” the external environment, a set of actuator functions, or actions that can be taken, a fitness function that measures the success of the last action, and a set of strengths that are randomly selected to perform the next action (Figure 5).

The strengths in the table are adjusted by the fitness function that measured how well a decision moved the system as a whole towards the desired end (maximization or minimization of the fitness function, depending on the application). In the beginning, all decisions are equally probable (as reflected by equal strengths in the table), but over time, the fitness function adjusts the strengths in the table and certain decisions become more probable in context.

3.7.5 XML Processor Classes

The XML Processor class (which includes Characters, Processor, and XMLTable) provides simulations with a W3C recommendations-compliant, minimal non-validating XML processor with a

\(^\text{11}\)This GA does a good job of letting sellers of monopolist companies find the greatest price the market will bear, as well as letting sellers in competitive markets drive down prices. It does less well in oligopolistic markets, where game theoretic strategies are more successful.
SAX-like interface. For example, the ModelGen class uses the XML Processor to read input files and create a simulation.

The XML Processor works by (1) applying events to the XML document being read and (2) defining call-back functions for those events. For example, when the Processor encounters a start tag in an XML document, it considers the encounter an event and notifies the Model object by calling a start-tag call-back function.

The Model implements the AgentLib XML Processor by providing call-back functions customized to the particular document it expects to read (e.g., the input specifications file used to create a simulation) and then reacting appropriately to the sequence of events. Because of this flexibility, the XML Processor can be reused for different kinds of documents simply by providing different sets of call-back functions (conveniently organized by deriving from, and overriding virtual functions in, the XMLEventHandler class).

4 N-ABLE Classes

The set of classes that constitute the N-ABLE simulation engine are built largely using classes from the AgentLib library. For example, N-ABLE simulations are controlled by the NABLEModel class, which inherits from AgentLib’s Model class. The fundamental N-ABLE economic agent is the EconomicAgent, which inherits from the AgentLib Agent class. Other important N-ABLE classes include Market(which contains the base-level Seller and Buyer classes), Infrastructure, Shipper, and Router. The N-ABLE classes are discussed in turn.

4.1 EconomicAgent

The EconomicAgent (Figure 7) is the primary class for modelling the buying, selling, production, economic planning, and learning of economic agents in an N-ABLE economy; this includes manufacturing and service firms, households, banks, and government institutions. Starting from the center of the figure, economic agents typically conduct one or more productive activities, whether it be the consumption of goods (e.g., by households), the production of goods and services (e.g., by

\[12\] In economics, households consume goods so as to produce “utility,” or personal satisfaction/contentment.
manufacturing firms), or the production and distribution of utilities (e.g., by electric power utilities and transmission/distribution infrastructure). When a Production runs out of inputs, Buyers on the left go to Markets and buy the goods; when Production has marketable outputs, Sellers on the right sell the goods in Markets.

For each of production, there is one seller for each market in which this firm’s good can be sold. Each production typically has one or more commodity inputs (e.g., producing a car requires metal, tires, glass, electronics, and more). The EconomicAgent assigns one buyer to each input material, regardless of the number of productions in which the material is used. This buyer periodically checks the inventory of this material and when the material is low, he goes to its market and attempts to purchase more. (Each buyer and seller has a unique AgentLib ObjectID and knows their parent EconomicAgent’s unique AgentID so that Message Events can be sent between them for the purposes of buying and selling.

Sellers, production, and buyers work independently but cooperatively to keep the firm productive and profitable. When a new order comes in to a seller, it attempts to sell product in the Warehouse; if there is not enough, the seller denies the order but then tells production to make more of the product. Buyers, in addition to periodically checking inventory, also check to see if a lack of their input caused production to be stopped; if so, the buyer purchases more for inventory so as to prevent this “starvation” of production.

The Accountant keeps track of all order-related information in its accounts payable and accounts receivable records. When the firm gets a new order, it logs it in the accounts receivable record. When payment is received, the Accountant deposits the payment in the Bank. Conversely, when a buyer makes a purchase, the buyer’s Accountant logs the new order in its accounts payable records, and when the order is shipped and received, the buyer’s Accountant sends payment to the firm.

---

13 This is a constant returns to scale production technology: the fractional content of inputs is constant regardless of the production level. A decreasing returns to scale technology uses increasing fractions of inputs per unit output (it is less efficient as production increases), while increasing returns to scale uses decreasing fractions (is more efficient).
from which it purchased the product.

The Warehouse class stores all inventories of input and output commodities, whose types are selected from an enumeration list in the CommodityType class. The EconomicAgent also has a Wall object, which contains the Spigots from which Infrastructure class-based commodities are drawn (think of a spigot that taps into the water distribution) and the Sinks to which new infrastructure product is sent (think of a sink that pours water into a water distribution system, which then distributes the water to users who have Spigots).

Each EconomicAgent has indirect access to the NABLEModel Shipper, which ships discrete goods from one EconomicAgent to another. When a Seller within an EconomicAgent ships a product, it sends a shipment request to the Shipper who then picks up the package and delivers it to its destination. Finally, the EconomicAgent has a CommTerminal object, which delivers all electronic-based communications between agents, via a Router. When making these communications, agents within EconomicAgent can access the terminal and send a message to a Router object, which then forwards the message to the CommTerminal of the intended recipient. Currently, the Accountant uses the CommTerminal to make all financial transactions with the Bank (which is required by all N-ABLE simulations). Future applications within N-ABLE could use the CommTerminal and Router classes to model telephone calls (digital and analog), internet messaging, and electronic payments systems.

Broadly, the three N-ABLE infrastructure-related classes — Infrastructure, Shipper, and Router — model three quite different types of “real world” infrastructure systems. The Infrastructure class models the uni-directional conveyance of a continuous medium (e.g., water, natural gas, electric power) that has no properties other than a type declarer (such as “WATER”) and the ability to be used in very small, divisible units. Shippers, on the other hand, are designed to bi-directionally carry bulk commodity in fixed-size units (e.g., a package containing an order). CommTerminals are designed to bi-directionally send discrete data with potentially highly embodied intelligence (such as a list of payments for a bank to make, the contents of a URL, or specific order details).

---

14 This CommodityType class currently lists commodities such as “GOODS” and “CAPITAL.” Future versions of this class will allow for arbitrary designations of commodities and have an extendible list of commodity properties.

15 Specifically, the Seller schedules a “Ship” Message Event in the central Model Calendar, which then delivers the Message to Shipper. The Shipper then schedules a Task Event in the Model Calendar to ship and deliver to the Buyer in the customer EconomicAgent.
4.2 Firm and Its Subclasses

So as to provide a modular means for creating different economic agents that use different markets and associated contracts, the EconomicAgent is a flexible, extensible framework that can build different economic agents and market participants from building blocks. The Firm, InfraFirm, TransportFirm, CommFirm, and BankFirm classes provide those building blocks, by each (other than Firm) providing variants of the Firm class's FirmBuyer, FirmProduction, and FirmSeller agents, and the contracts that result between Buyers and Sellers.\(^\text{16}\)

For example, the Firm class includes FirmBuyer, FirmProduction, and FirmSeller classes which are designed for ordering, producing, and selling physical goods (e.g., that would be shipped by truck). In contrast, the InfraFirm class, a subclass of Firm, includes InfraBuyer, InfraProduction, and InfraSeller classes which know how to buy an infrastructure commodity (by buying a Spigot), how to produce an infrastructure commodity (by producing and “pouring” into a Sink), and how to sell an infrastructure commodity (by selling a Spigot), respectively.

Figure 10 shows how these Firm classes and sub-classes are used to create buyers, productions, and sellers in an economic firm. The firm in the figure uses three inputs for production: a banking input (a bank account for handling payments and receipts), an infrastructure input (POWER), and a material input (FABMETAL). The buyer for the banking input is constructed as a BankFirmBuyer, located in the BankFirm class; the buyer for the infrastructure input is constructed as InfraFirmBuyer, located in the InfraFirm class, and the buyer of goods is constructed as a FirmBuyer, from the Firm class. Since the firm is producing a physical good ("GOODS"), its production is created as a FirmProduction object, and it is sold using a FirmSeller, also from the Firm class.\(^\text{17}\)

By specifying all of these objects in the N-ABLE simulation input file, each EconomicAgent can properly purchase inputs, produce, and sell in markets.

\(^{16}\)FirmBuyer and FirmSeller inherit from the Market class’s Buyer and Seller, and FirmProduction inherits from the Production class.

\(^{17}\)To be clear, GOODS and all other commodities in the CommodityType class are arbitrary designations, that is, the name of the commodity does not imply any restriction on how the commodity can be produced, bought, or sold. POWER could be sold in discrete units via the Firm class, and FABMETAL could be sold in divisible units via the InfraFirm class. It would just be confusing.
4.3 Economic Markets and Infrastructure Classes

N-ABLE economic agents communicate and contract with one another via a number of markets and infrastructure.

4.3.1 Economic Markets

The Market class (which derives from AgentLib BBoard) is used to create markets (or BBoard directories) for Buyers and Sellers to find one another and make economic exchanges of goods for money. When N-ABLE first instantiates agents at the beginning of a simulation, each Buyer created for a firm has a designated Region, such as “Region1,” which defines a spatial or contextual region for the market. The Buyer also has a CommodityType-based commodity it will be purchasing; the combination of CommodityType and Region uniquely defines this market. When the Buyer is created, the Model creates this Market (unless it has been already created for a previously created Buyer or Seller) and the Buyer is then added to this Market/BBoard listing. When all Region1 buyers and sellers have been created, the commodity-Region1 paired Market object contains a directory of all of its Buyers and Sellers. This directory is the means by which a Buyer can locate a Seller of the commodity.
Buyers and sellers can have varying rules of exchange, depending on the type of market. Buyers and sellers that derive from the N-ABLE “Firm” class purchase and sell goods in discrete chunks, with maximum acceptable prices and immediate billing. In contrast, InfraBuyers and InfraSellers in the InfraMarket class purchase and sell Spigots that give access to the infrastructure commodity, and bill monthly. In both cases, the default market mechanism for Buyers finding the best seller is a sequential search algorithm: the buyer first selects a potential seller and asks for its price for the good or service. If the buyer determines that it may get a lower price if it keeps shopping, it will. The buyer continues this sequential searching until the expected benefit of an additional search is no longer greater than the cost of this search.\textsuperscript{18}

### 4.3.2 Infrastructure

The Infrastructure class is used by NABLEModel to handle the distribution of water, electric power, and other public utility-type commodities from producers to users. The Infrastructure object has the functionality to accept new commodity into its system (though the Sinks on Producers’ Walls), to distribute the commodity to users (through Spigot objects on the users’ Walls), and to ration the distribution when demand for the commodity exceeds supply (through an interrupt detector in the Infrastructure object). Figure 12 illustrates how Infrastructure Sinks and Spigots provide access to distribution network capabilities.

Each InfraBuyer periodically decides whether to shop around for another, cheaper provider of the commodity; if it does, then it removes its existing Spigot, pays the remaining balance on its existing contract and closes the contract, and then attaches the new Spigot to its Wall.

### 5 Examples of N-ABLE Economic Firms

The following examples illustrate how the AgentLib and N-ABLE classes are used to make economic firms. Each is a variant of the basic EconomicAgent structure of production, sellers, buyers, and economic accounting. Each also uses a different combination of messaging between agents within an EconomicAgent firm and between EconomicAgents.

\textsuperscript{18}For more details on this search rule, see Carlson and McAfee\textsuperscript{[5]} and Ehlen\textsuperscript{[6]}. 
5.1 Manufacturers

Figure 13 illustrates a typical N-ABLE manufacturing firm (this firm is similar to the EconomicAgent shown in Figure 7). The firm uses three inputs to produce two goods. In-house inventories of input and output goods are stored in the EconomicAgent Warehouse.

Following the life-cycle of an order, the sellers update daily their prices, using price classifiers (derived from AgentLib’s Classifier class) to “learn” from past pricing to find the best prices for this firm in their markets. When a seller receives an order, the Accountant posts the order in its accounts receivable log. If the seller can’t sell from inventory in the Warehouse, he increases production. As production occurs, the material inputs are used in constant proportions to their output “recipe” which specifies the quantity of each input is required per unit output. When the order is filled, the seller contacts the Shipper and requests that the product be shipped to the customer buyer agent. Ultimately, the Accountant gets a payment from the buyer’s Accountant and then deposits it in the Bank, which then debits the Buyers account and credits the seller’s account.\(^{19}\)

5.2 Consumers

Figure 14 illustrates a typical consumer. This consumer uses three goods to “produce” two different “utilities.” Since consumers do not in general sell this utility (they are consumers), the consumer EconomicAgent does not have Seller agents, does not ship goods, and keeps no entries in its accounts receivable log. Otherwise, it is identical in function to a manufacturing firm.\(^{20}\) Additional example constructions of EconomicAgent-based infrastructure providers and banks could be displayed as

\(^{19}\)If the buyer and seller use different banks, then these banks handle the transaction with an interbank payment.

\(^{20}\)The next version of N-ABLE will have markets for labor, so that households will work at firms and receive income. N-ABLE will then model circular economies where the flow of goods and services (including labor) flow in one direction and money flows in the opposite direction.
6 Examples of N-ABLE Simulations

The following three example simulations illustrate how N-ABLE classes-based agents are used to create simulations. Each is intentionally simple in nature; large and elaborate simulations are built as extensions from these types of sub-assemblies. All N-ABLE simulations are essentially data-driven: all of the particular settings for the number and types of agents and infrastructure are set in the input file. No N-ABLE or AgentLib classes need to be modified.

6.1 Single Economic Market

Figure 15 illustrates a single economic market composed of five firms all selling a single good to numerous consumers. The FirmBuyer, FirmSeller, and FirmProduction classes are used so that consumers can buy from the firms; the Bank and Router classes provide the means for making payments and storing money; and the Shipper class provides the means for delivering the goods to households. Demand for the good, delivery of goods, and payment happen as follows. Each time period, the N-ABLE Consumer class produces “utility” by consuming the input goods from inventory. Each FirmBuyer agent assigned to each input checks inventory of the good and if low, goes to the FirmMarket to purchase. Using the sequential search algorithm, the FirmBuyer selects the first FirmSeller who has the amount of good needed and at a low enough price as determined by the search algorithm. If successful, the FirmBuyer makes a purchase and the FirmSeller contacts the Shipper to deliver the goods. Once delivered, the FirmBuyer sends a payment to the FirmSeller, who then uses the CommTerminal and Router to make payment to the Bank.
6.2 Four-Stage Supply Chain

Figure 16 illustrates a four-step supply chain, where firms produce a good in sequence. Consumers consume the Final Good and when their inventories are low, purchase from Firm 3. FirmBuyers, FirmSellers, FirmMarkets, Shipping, and banking are mechanically the same as in the previous example. When Firm 3 runs out of the Intermediate Good material it needs to produce the Final Good, it purchases more from Firm 2. Analogously, when Firm 2 runs out of Raw Materials it purchases more from Firm 1.

Over the long term, consumer demand “pulls” goods through the system, and the ability of the supply chain to provide to consumers is determined ultimately by the amount of Raw Material that Firm 1 has in inventory. In the short term, the dynamics of supply, demand, and inventories are quite complex.\textsuperscript{21}

\textsuperscript{21}This is a variant of the \textit{beer game} model, a supply chain that displays chaotic output and inventory behavior.
6.3 Infrastructure Market

Figure 17 illustrates a basic market where one firm sells water to two households. The InfraFirm-Buyer, InfraFirmSeller, InfraFirmMarke provide the means for households can shop for and purchase a utility contract and then use water. InfraFirmProduction provides the means for the firm can produce water and supply it to the water distribution system. At the beginning of a simulation, the households draw water from their own inventories. When inventories get low, the household’s InfraFirmBuyers go to the InfraFirmMarket and buy a spigot from the Water Firm. Once purchased and installed, the households draw water as needed. Each time period the Water Firm’s InfraFirmProduction checks its Sink to see what the “pull” for water is and then produces to that pull. Each month, the Water Firm’s Accountant bills the water users, who then pay via the Bank.

7 Overview of User Interface

The N-ABLE user interface is a Windows-based framework for organizing the data for input, for submitting input files to N-ABLE, for real-time monitoring of data, and for viewing and archiving simulation output. The following screens, which are likely to be the only interaction most analysts have with N-ABLE, briefly illustrate how N-ABLE carries out some of these tasks.

Figure 18 shows the initial screen, displays a navigation panel for new users, and a list simulations either currently running on the SimRunner, archiving data on the SimStreamer, or both. Figure 19 displays the screen used to construct simulations from constituent agents and objects (e.g., EconomicAgent variants and infrastructure). On the left-hand side is a treeview listing the current set of agents, their components, and new branches for creating new agents or components. The right hand panel is used to input and edit the object selected in the tree. The N-ABLE user interface has three main types of results screens: timeseries, 2D scatter diagrams, 3D point clouds. Figure 20 displays the timeseries graph; this figure along with the three others all have the left-hand-side tree view, which contains a hierarchy of all snapshot data, by simulation, agent type, (AgentLib) AgentID, and then data element. For example, Figure 20 shows three timeseries of data for Firm agent: advertised price, sold amount, and profits.

See North et al[10] for a description and comparison of alternate implementations.

22 The N-ABLE 2.0 interface will be Java-based.
Figure 18: User Interface - Simulation Console

Figure 19: User Interface - Inputs
Figure 20: User Interface - Timeline Results Data
8 Summary

The N-ABLE architecture is a collection of objects that represent economic agents, markets, and infrastructure. In addition to traditional microeconomic and macroeconomic constructs, these objects help model the physics of how the economics is carried out: how buyers meet sellers in markets, how physical discrete commodities are transported, how flow commodities like electric power are established and rationalized during outage, and telephony and other data travel or breakdown over data networks.

N-ABLE accomplishes this through two distinct sets of classes. The first set of classes, AgentLib, provides the basic structure necessary to instantiate new agents, schedule their daily, weekly, and monthly activities, and send messages between them (particularly when the agents reside on different nodes in massively parallel systems). To insure that these classes work well for a range of applications, there is no economic or other subject-specific code in AgentLib. The second set of classes are specific N-ABLE classes that allow for the modelling of economic markets, economic production within firms, economic accounting, physical product shipment, utility distribution, and more.

N-ABLE agents are built by assembling agents and other objects from the AgentLib library. Specific economic agents and supporting infrastructure are build from EconomicAgents, infrastructure objects, and other N-ABLE objects. Due to the inherent data-driven design of the AgentLib library, a wide range of N-ABLE simulations can be constructed and run.

References


**Distribution**

<table>
<thead>
<tr>
<th>Number</th>
<th>Code</th>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0161</td>
<td>Patent &amp; Licensing Office</td>
<td>11500</td>
</tr>
<tr>
<td>1</td>
<td>0619</td>
<td>Review &amp; Approval Desk</td>
<td>09216 for DOE/OSTI</td>
</tr>
<tr>
<td>2</td>
<td>0899</td>
<td>Technical Library</td>
<td>09616</td>
</tr>
<tr>
<td>1</td>
<td>9018</td>
<td>Central Technical Files</td>
<td>08945-1</td>
</tr>
<tr>
<td>1</td>
<td>0741</td>
<td>M. Tatro</td>
<td>06200</td>
</tr>
<tr>
<td>1</td>
<td>1138</td>
<td>J. L. Mitchiner</td>
<td>06220</td>
</tr>
<tr>
<td>1</td>
<td>1138</td>
<td>D. Horschel</td>
<td>06226</td>
</tr>
<tr>
<td>10</td>
<td>1138</td>
<td>M. A. Ehlen</td>
<td>06221</td>
</tr>
<tr>
<td>10</td>
<td>1138</td>
<td>E. D. Eidson</td>
<td>06221</td>
</tr>
<tr>
<td>1</td>
<td>1138</td>
<td>A. J. Scholand</td>
<td>06221</td>
</tr>
<tr>
<td>1</td>
<td>1138</td>
<td>P. S. Downes</td>
<td>06221</td>
</tr>
<tr>
<td>1</td>
<td>1138</td>
<td>D. K. Belasich</td>
<td>06221</td>
</tr>
<tr>
<td>1</td>
<td>0318</td>
<td>D. A. Schoenwald</td>
<td>09216</td>
</tr>
<tr>
<td>1</td>
<td>0196</td>
<td>B. N. Chenowethe Creel</td>
<td>09216</td>
</tr>
<tr>
<td>1</td>
<td>0318</td>
<td>J. A. Sprigg</td>
<td>09216</td>
</tr>
</tbody>
</table>