A Software Development Framework for Agent-Based Infectious Disease Modelling

Chapter · January 2011

DOI: 10.5772/13669 · Source: InTech

4 authors, including:

Luiz C Mostaço-Guidolin
University of Manitoba
11 PUBLICATIONS **65** CITATIONS
SEE PROFILE

Nick Pizzi
The University of Winnipeg
73 PUBLICATIONS **598** CITATIONS
SEE PROFILE

33

A Software Development Framework for Agent-Based Infectious Disease Modelling

Luiz C. Mostaço-Guidolin[1,2], Nick J. Pizzi[2,3],
Aleksander B. Demko[2,3] and Seyed M. Moghadas[1,2]
[1]*Department of Mathematics and Statistics, University of Winnipeg,*
[2]*Institute for Biodiagnostics, National Research Council Canada,*
[3]*Department of Computer Science, University of Manitoba,*
Canada

1. Introduction

From the Black Death of 1347–1350 (Murray, 2007) and the Spanish influenza pandemic of 1918–1919 (Taubenberger & Morens, 2006), to the more recent 2003 SARS outbreak (Lingappa et al., 2004) and the 2009 influenza pandemic (Moghadas et al., 2009), as well as countless outbreaks of childhood infections, infectious diseases have been the bane of humanity throughout its existence causing significant morbidity, mortality, and socioeconomic upheaval. Advanced modelling technologies, which incorporate the most current knowledge of virology, immunology, epidemiology, vaccines, antiviral drugs, and public health, have recently come to the fore in identifying effective disease mitigation strategies, and are being increasingly used by public health experts in the study of both epidemiology and pathogenesis. Tracing its historical roots from the pioneering work of Daniel Bernoulli on smallpox (Bernoulli, 1760) to the classical compartmental approach of Kermack and McKendrick (Kermack & McKendrick, 1927), modelling has evolved to deal with data that is more heterogeneous, less coarse (based at a community or individual level), and more complex (joint spatial, temporal and behavioural interactions). This evolution is typified by the agent-based model (ABM) paradigm, lattice-distributed collections of autonomous decision-making entities (agents), the interactions of which unveil the dynamics and emergent properties of the infectious disease outbreak under investigation. The flexibility of ABMs permits an effective representation of the complementary interactions between individuals characterized by localized properties and populations at a global level.

However, with flexibility comes complexity; hence, the software implementation of an ABM demands more stringent software design requirements than conventional (and simpler) models of the spread and control of infectious diseases, especially with respect to outcome reproducibility, error detection and system management. Outcome reproducibility is a challenge because emergent properties are not analytically tractable, which is further exacerbated by subtle and difficult to detect errors in algorithm logic and software design. System management of software simulating populations/individuals and biological /physical interactions is a serious challenge, as the implementation will involve distributed (parallelized), non-linear, complex, and multiple processes operating in concert. Given these

issues, it is clear that any implementation of an ABM must satisfy three objectives: reliability, efficiency, and adaptability. Reliability entails robustness, reproducibility, and validity of generated results with given initial conditions. Efficiency is essential for running numerous experiments (simulations) in a timely fashion. Adaptability is also a necessary requirement in order to adjust an ABM system as changes to fundamental knowledge occur. Past software engineering experience (Pizzi & Pedrycz, 2008; Pizzi, 2008) and recent literature (Grimm & Railsback, 2005; Ormerod & Rosewell, 2009; Railsback et al., 2006; Ropella et al., 2002) suggest several guidelines to which ABM development should adhere. These include:

i. *Spiral methodology*. ABM software systems require rapid development, with continual changes to user requirements and incremental improvements to a series of testable prototypes. This demands a spiral methodology for software development, beginning with an initial prototype and ending with a mature ABM software release, via an incremental and iterative succession of refined requirements, design, implementation, and validation phases.
ii. *Activity streams*. Three parallel and complementary activity streams (conceptual, algorithmic, and integration) will be required during the entire software development life cycle. High-level analytical ABM concepts drive the creation of functionally relevant algorithms, which are implemented and tested, and, if validated, integrated into the existing code base. Normally considered a top-down approach, in a spiral methodology, bottom-up considerations are also relevant. For instance, the choice from a set of competing conceptual representations for an ABM model may be made based on an analysis of the underlying algorithms or the performance of the respective implementations.
iii. *Version control*. With a spiral development methodology, an industry standard version control strategy must be in place to carefully audit changes made to the software (including changes in relation to rationales, architects, and dates).
iv. *Code review*. As code is integrated into the ABM system, critical software reviews should be conducted on a regular basis to ensure that the software implementation correctly captures the functionality and intent of the over-arching ABM.
v. *Validation*. A strategy must be established to routinely and frequently test the software system for logic and design errors. For instance, the behaviour of the simulation model could be verified by comparing its output with known analytical results for large-scale networks. Software validation must be relevant and pervasive across guidelines (i)–(iv).
vi. *Standardized software development tools*. Mathematical programming environments such as Matlab® (Sigmon & Davis, 2010), Mathematica® (Wolfram, 1999), and Maple® (Geddes et al., 2008) are excellent development tools for rapidly building ABM prototypes. However, performance issues arise as prototypes grow in size and complexity to become software systems. A development framework needs to provide a convenient bridge from these prototyping tools to mature efficient ABM systems.
vii. *System determinism*. In a parallel or distributed environment, outcome reproducibility is difficult to achieve with systems comprising stochastic components. Nevertheless, system determinism is a requirement even if executed on a different computer cluster.
viii. *System profiling*. It is important to observe and assess the performance of parts of the system as it is running. For instance, which components are executed often; what are their execution times; are processing loads balanced across nodes in a computer cluster?

In order to adhere to these guidelines and satisfy the objectives described above, we designed a software development framework for ABMs of infectious diseases. The next section of this chapter describes Scopira, a general application development library designed by our research group to be a highly extensible application programming interface with a wholly embedded transport layer that is fully scalable from single machines to site-wide distributed clusters. This library was used to implement the agent-based modelling framework, details of which are provided in the subsequent section. We conclude with a section describing future research activities.

2. Scopira

In the broad domain of biomedical data analysis applications, preliminary prototype software solutions are usually developed using an interpreted programming language or environment (e.g., Matlab®). When performance becomes an issue, some components of the prototype are subsequently ported to a compiled language (e.g., C) and integrated into the remaining interpreted components. Unfortunately, this process can introduce logic and design errors and the functionality of resultant hybrid system can often be difficult to extend or adapt. Further, it also becomes difficult to take advantage of features such as memory management, object orientation, and generics, which are all essential requirements for building large scale, robust applications. To address these concerns, we developed Scopira (Demko & Pizzi, 2009), an open source C++ framework suitable for biomedical data analysis applications such as ABMs for infectious diseases. Scopira provides high performance end-to-end application development features, in the form of an extensible C++ library. This library provides general programming utilities, numerical matrices and algorithms, parallelization facilities, and graphical user interface elements.

Our motivation behind the design of Scopira was to satisfy the needs of three categories of users within the biomedical research community: software architects; scientists /mathematicians; and data analysts. With the design and implementation of new software, architects typically need to incorporate legacy systems often written in interpreted languages. Coupled with the facts that end-user requirements in a research environment often change (sometimes radically) and that biomedical data is becoming ever more complex and voluminous, a software development framework must be versatile, extensible, and exploit distributed, generic, and object oriented programming paradigms. For scientists or mathematicians, data analysis tools must be intuitive with responsive interfaces that operate both effectively and efficiently. Finally, the data analyst has requirements straddling those from the other user categories. With an intermediate level of programming competence, they require a relatively intuitive development environment that can hide some of the low level programming details, while at the same time allowing them to easily set up and conduct numerical experiments that involve parameter tuning and high-level looping/decision constructs. As a result of this motivation, the emphasis with Scopira has been on high performance, open source development and the ability to easily integrate other C/C++ libraries used in the biomedical data analysis field by providing a common object-oriented application programming interface (API) for applications. This library provides a large breadth of services that fall into the following four component categories.

Scopira Tools provide extensive programming utilities and idioms useful to all application types. This category contains a reference counted memory management system, flexible/redirectable flow input/output system, which supports files, file memory mapping,

network communication, object serialization and persistence, universally unique identifiers (UUIDs) and XML parsing and processing.
The *Numerical Functions* all build upon the core n-dimensional *narray* concept (see below). C++ generic programming is used to build custom, high-performance arrays of any data type and dimension. General mathematical functions build upon the *narray*. A large suite of biomedical data analysis and pattern recognition functions is also available to the developer.
Multiple APIs for *Parallel Processing* are provided with the object-oriented framework, *Scopira Agents Library* (SAL)‡, which allows algorithms to scale with available processor and cluster resources. Scopira provides easy integration with native operating system threads as well as the Message Passing Interface (MPI) (Snir & Gropp, 1998) and Parallel Virtual Machine (PVM) (Geist et al., 1994) libraries. Further, this library may be embedded into desktop applications allowing them to use computational clusters automatically, when detected. Unlike other parallel programming interfaces such as MPI and PVM, Scopira's facilities provide an object-oriented strategy with support for common parallel programming patterns and approaches.
Finally, a *Graphical User Interface* (GUI) *Library* based on GTK+ (Krause, 2007) is provided. This library provides a collection of useful widgets including a scalable numeric matrix editor, plotters, image and viewers as well as a plug-in platform and a 3D canvas based on OpenGL® (Hill & Kelley, 2006). Scopira also provides integration classes with the popular Qt GUI Library (Summerfield, 2010).

2.1 Programming utilities

Intrusive reference counting (recording an object's reference count within the object itself) provides the basis for memory management within Scopira-based applications. Unlike many referencing counting systems (such as those in VTK (Kitware, 2010) and GTK+), Scopira's system uses a decisively symmetric concept. References are only added through the *add_ref* and *sub_ref* calls — specifically, the object itself is created with a reference count of zero. This greatly simplifies the implementation of smart pointers and easily allows stack allocated use (by passing the reference count), unlike VTK and GTK+ where objects are created with a reference count and a modified reference count, respectively. Scopira implements a template class *count_ptr* that emulates standard pointer semantics while providing implicit reference counting on any target object. With smart pointers, reference management becomes considerably easier and safe, a significant improvement over C's manual memory management.
Scopira provides a flexible, polymorphic and layered input/output system. Flow objects may be linked dynamically to form I/O streams. Scopira includes *end flow* objects, which terminate or initiate a data flow for standard files, network sockets and memory buffers. *Transform flow* objects perform data translation from one form to another (e.g., binary-to-hex), buffer consolidation and ASCII encoding. *Serialization flow* objects provide an interface for objects to encode their data into a persistent stream. Through this interface, large complex objects can quickly and easily encode themselves to disk or over a network. Upon reconstruction, the serialization system re-instantiates objects from type information stored in the stream. Shared objects — objects that have multiple references — are serialized just once and properly linked to multiple references.

‡ The term "agent" used in this context refers to the software concept rather than the modelling concept. To avoid confusion, we will use the term "SAL node" to refer to the software concept.

A platform independent configuration system is supplied via a central parsing class, which accepts input from a variety of sources (e.g., configuration files and command line parameters) and present them to the programmer in one consistent interface. The programmer may also store settings and other options via this interface, as well as build GUIs to aid in their manipulation by the end user. Using a combination of the serialization type registration system and C++'s native RTTI functions, Scopira is able to dynamically (at runtime) allow for the registration and inspection of object types and their class hierarchy relationships. From this, an application plug-in system can be built by allowing external dynamic link libraries to register their own types as being compatible with an application, providing a platform for third party application extensions.

2.2 N-dimensional data arrays

The C and C++ languages provide the most basic support for one-dimensional arrays, which are closely related to C's pointers. Although usable for numerical computing, they do not provide the additional functionality that scientists and mathematicians demand such as easy memory management, mathematical operations, or fundamental features such as storing their own dimensions. Multiple dimensional arrays are even less used in C as they require compile-time dimension specifications, drastically limiting their flexibility. The C++ language, rather than design a new numeric array type, provides all the necessary language features for developing such an array in a library. Generic programming (via C++ templates, that allow code to be used for any data types at compile time), operator overloading (e.g., being able to redefine the addition "+" or assignment "=" operators) and inlining (for performance) provide all the tools necessary to build a high performance, usable array class. Rather than force the developer to add another dependent library for an array class, Scopira provides n-dimensional arrays through its *narray* class. This class takes a straightforward approach, implementing n-dimensional arrays, as any C programmer would have, but providing a type safe, templated interface to reduce programming errors and code complexity. The internals are easy to understand, and the class works well with standard C++ library iterators as well as C arrays, minimizing lock-in and maximizing code integration opportunities. Using basic C++ template programming, we can see the core implementation ideas in the following code snippet:

```
template <class T, int DIM> class narray {
T* dm_ary;                    // actual array elements
nindex<DIM> dm_size;          // the size of each of the dimensions

T get(nindex<DIM> c) const {
                        assert(c<dm_size);
                        return dm_ary[dm_size.offset(c)];
        }
}
```

From this code snippet we can see that an *narray* is a template class with two compile time parameters: *T*, the element data type (*int*, *float*, etc.) and *DIM*, the number of dimensions. The actual elements are stored in a dynamically allocated C array, *dm_ary*. The dimension lengths are stored in an *nindex* type, a generic class at is used to store array offsets. A generalized accessor is provided, which uses the *nindex*-offset method to convert the dimension specific index and size of the array into an offset into the C array. This generalization works for any dimension size.

Another feature shown here is the use of C's *assert* macro to check the validity of the supplied index. This boundary check verifies that the index is indeed valid otherwise failing and terminating the program while alerting the user. This check greatly helps the programmer during the development and testing stages of the application, and during a high performance/optimized build of the application, these macros are transparently removed, obviating any performance penalties from the final, deployed code. More user-friendly accessors (such as those taking an *x* value or an *x-y* value directly) are also provided. Finally, C++'s operator overloading facilities are used to override the bracket "[]" and parenthesis "()" operators to give the arrays a more succinct feel, over explicit *get* and *set* method calls.

The *nslice* template class is a virtual n-dimensional array that is a *reference* to an *narray*. The class only contains dimension specification information and is copyable and passable as function parameters. Element access translates directly to element accesses in the host *narray*. An *nslice* must always be of the same numerical type as its host *narray*, but can have any dimensionality less than or equal to the host. This provides significant flexibility; one could have a one-dimensional vector slice from a matrix, cube or five-dimensional array, for example. Matrix slices from volumes are quite common (see Figure 1). These sub slices can also span any of the dimensions/axes, something not possible with simple pointer arrays (e.g., matrix slices from a cube array need not follow the natural memory layout order of the array structure).

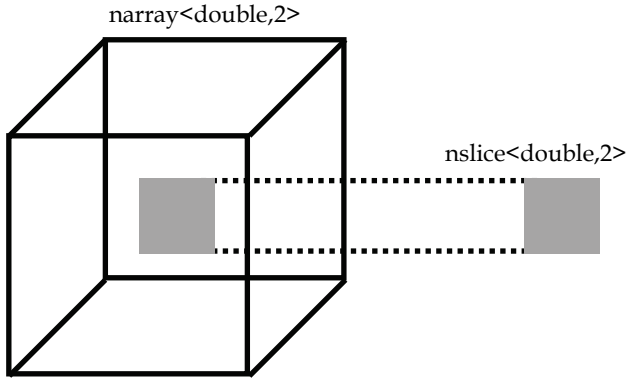

Fig. 1. An *nslice* reference into an *narray* data item.

The *narray* class provides hooks for alternate memory allocation systems. One such system is the *DirectIO* mapping system. Using the memory mapping facilities of the operating system (typically via the *mmap* function on POSIX systems), a disk file may be *mapped* into memory. When this memory space is accessed, the pages of the files are loaded into memory transparently. Writes to the memory region will result in writes to the file. This allows files to be loaded in portions and on demand. The operating system will take care of loading/unloading the portions as needed. Files larger than the system's memory size can also be loaded (the operating system will keep only the working set portion of the array in memory). The programmer must be aware of this and take care to keep the working set within the memory size of the machine. If the working set exceeds the available memory size, performance will suffer greatly as the operating system pages portions to and from disk (this excessive juggling of disk-memory mapping is called "page thrashing").

2.3 Parallel processing

With the increasing number of processors in both the user's desktops and in cluster server rooms, computationally intensive applications and algorithms should be designed in a parallel fashion if they are to be relevant in a future that depends on multiple-core and cluster computing as a means of scaling processing performance. To take advantage of the various processors within a single system or shared address space (SAS), developers need only utilize the operating system's thread API or shared memory services. However, for applications that would also like to utilize the cluster resources to achieve greater scalability, explicit message passing is used. Although applying a SAS model to cluster computing is feasible, to achieve the best computational performance and scalability results, a message passing model is preferred (Shan et al., 2003; Dongarra & Dunigan, 1997). Scopira includes support for two well established message passing interfaces, MPI and PVM, as well as a custom, embedded, object-oriented message passing interface designed for ease of use and deployment.

SAL is a parallel execution framework extension with several notable goals particularly useful to Scopira based applications. The API, which is completely object-oriented, includes functionality for: using the flow system for messaging; task movement; GUI application integration; multi-platform communication support; and, the registration system for task instantiation. SAL introduces high-performance computing to a wider audience of end users by permitting software developers to build standard cluster capabilities into desktop applications, allowing those applications to pool their own as well as cluster resources. This is in contrast to the goals of MPI (providing a dedicated and fast communications API standard for computer clusters) and PVM (providing a virtual machine architecture among a variety of powerful computer platforms).

By design, SAL borrowed a variety of concepts from both MPI and PVM. SAL, like PVM, attempts to a build a unified and scalable "task" management system with an emphasis on dynamic resource management and interoperability. Users develop intercommunicating task objects. Tasks can be thought of as single processes or processing instances, except that they are implemented as language objects and not operating system processes. A SAL node manages one or more tasks, and teams of nodes communicate with each other to form computational networks (see Figure 2). The tasks are coupled with a powerful message passing API inspired by MPI. Unlike PVM, SAL also focuses on ease-of-use: emphasizing automatic configuration detection and de-emphasizing the need for infrastructure processes. When no cluster or network computation resources are available, SAL uses operating system threads to enable multi-programming within a single operating system process and thereby embedding a complete message passing implementation within the application (greatly reducing deployment complexity). Applications always have an implementation of SAL available, regardless of the availability or access to cluster resources. Developers may always use the message passing interface, and their application will work with no configuration changes from both single machine desktop installations to complete parallel compute cluster deployments.

The mechanics and implementation of the SAL nodes and their load balancing system are built into the SAL library, and thereby, Scopira applications. Users do not need to install additional software, nor do they need to explicitly configure or set-up a parallel environment. This is paramount in making cluster and distributed computing accessible to the non-technical user, as it makes it a transparent feature in their graphical applications.

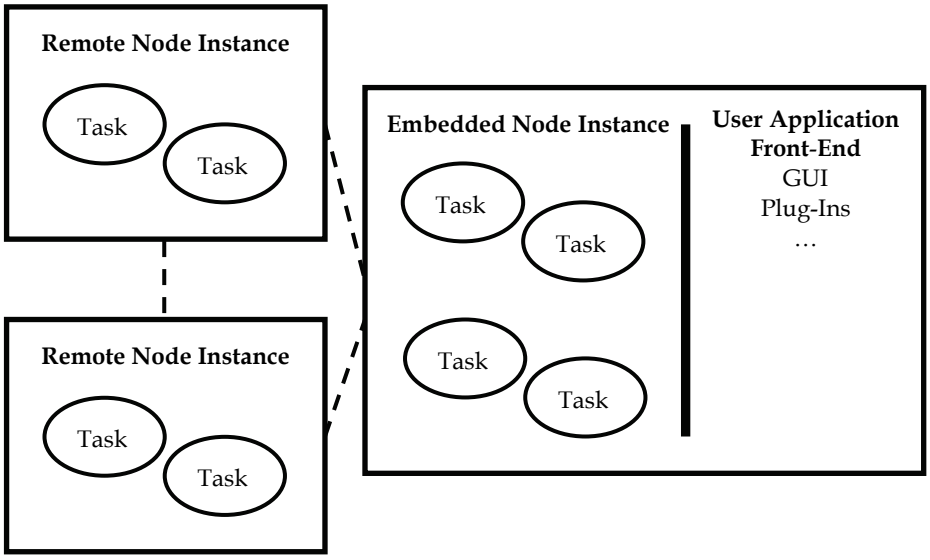

Fig. 2. SAL topology of tasks and nodes.

SAL provides an object-oriented, packet based and routable (like PVM, but unlike MPI) API for message passing. This API provides everything needed to build multi-threaded, cluster-aware algorithms embeddable in applications. Tasks are the core objects that developers build for the SAL system. A task represents a single job or instance in the SAL node system, which is analogous to a process in an operating system. However, they are almost never separate processes, but rather grouped into one or more SAL node processes that are embedded into the host application. This is unlike most existing parallel APIs, that allocate one operating system process per task concept, that, although conceptually simpler for the programmer, incurs more communication and start up overhead, as well as making task management more complex and operating system dependent. The tasks themselves are language-level objects but are usually assigned their own operating system threads to achieve pre-emptive concurrency.

A context object is a task's gateway into the SAL message passing system. There may be many tasks within one process, so each will have differing context interfaces – something not feasible with an API with a single, one-task-per-process model (as used in PVM or MPI). This class provides several facilities, including: task creation and monitoring; sending, checking and receiving messages; service registration; and group management. It is the core interface a developer must use to build parallel applications with SAL.

Developers often launch a group of instances of the same task time, and then systematically partition the problem space for parallel processing. To support this popular paradigm of development, SAL's identification system supports the concept of *groups*. A group is simply a collection of N task instances, where each instance has a *groupid*={0,…,N–1}. The group concept is analogous to MPI's communicators (but without support for complex topology) and PVM's named groups. This sequential numbering of task instances allows the developer to easily map problem work units to tasks. Similar to how PVM's group facility supplements the task identifier concept, SAL groups build upon the UUID system, as each task still functionally retains their underlying UUID for identification.

The messaging system within SAL is built upon both the generic Scopira input/output layer as well as the UUID identification system. SAL employs a packet-based (similar to PVM)

message system, where the system only sends and routes complete messages, and not the individual data primitives (as MPI can and routinely does) and objects within them. Only after the sending task completes and commits a message is it processed by the routing and delivery sub-systems. A node uses operating system threads to transport the data, freeing the user's thread to continue to work.

Sending (committing) the data during the *send_msg* object's destruction (that is, via its destructor) was the result of an intentional software design decision. In C++, stack objects are destroyed as they exit scope. The user should therefore place a *send_msg* object in its own set of scope-braces, which would constitute a sort of "send block". All data transmissions for the message would be done within that send block, and the software programmer can then be assured that the message will be sent at the end of the scope block without having to remember to do a manual send commit operation. Similarly, the receiver uses a *recv_msg* object to receive, decode and parse a message packet, all within a braced "receive block." The following code listing provides an example of a task object that, via its context interface (the interface to the message network), is sending a variety data objects using the object-oriented messaging API.

```
// declare my task object and its run method
class mytask : public agent_task_i
        { public: virtual void int run(task_context &ctx); };
int mytask::run(task_context &ctx) {
        // send some data to the master task
        narray<double,2> a_matrix;
        {                               // this scope (or send) block encapsulates the sent message
        send_msg msg(ctx, 0);       // prepare the message to task #0

        msg.write_int(10);// send one integer
        msg.save(a_matrix);         // send a matrix – type safe
        msg.save(user_object);      // send a user object – via serialization

        // at this point, msg's destructor will be called (automatically)
        // triggering the sending of the message
        }
}
```

SAL currently has two types of scheduling engines, a "local" engine that uses operating system threads on a single host machine and a "network" implementation that is able to utilize a network of workstations. The "local" engine is a basic multi-threaded implementation of the SAL API. It uses the operating system's threads to implement multiprocessing within the host application process. As this engine is contained within a single-process, it is the fastest and easiest to use for application development and debugging. The programmer may fully design and test their parallel algorithm and its messaging logic before moving to a multi-node deployment. The local engine is always available and requires no configuration from the user. Developers need not write a dedicated non-message passing versions of their algorithms simply to satisfy users that may not go to the trouble of deploying a Cluster. As the local engine provides as many worker threads as active tasks, it relies on the operating system's ability to manage threads within the processors. This works quite well when the number of tasks instantiated into the system is a function of the number of physical processors, as encouraged by the API.

The network engine implements the SAL API over a collection of machines connected by an IP-based network; typically Ethernet. The cluster can be a dedicated computer cluster and/or a collection of user workstations. The engine itself provides inter-node routing and

management, leaving the local scheduling decisions within each node up to a local-engine derived manager. A SAL network stack has two layers (see Figure 3). The lower transport layer contains the SAL nodes themselves (objects that manage all the tasks and administration on a single process) and their TCP/IP based links. Tasks can send messages to each other using their UUIDs, ignorant of the IP layer or the connection topology of the nodes themselves. For simplicity and efficiency, a SAL network (like PVM) has a master node residing on one process. This master node is responsible for the allocation, tracking and migration of all the tasks in the system. The network engine uses a combination of URL-like direct addressing and UDP/IP broadcast based auto-discovery in building the node network. The simplest and most popular sequence is to start an application in auto discovery mode. When a network engine starts, it searches the local network for any other node peers and, if found, joins their network. If no peers are found, then it starts a network consisting of itself as the only member and assumes the master node role. Users may also key in the master's URL directly, connecting them explicitly to a particular network. In addition to its critical routing functions, the master node is also responsible for all the task tracking and management within the network. By centralizing this information, load and resource allocation decisions can be made instantly and decisively. The master node handles all task instantiation requests. When a task within node requests the creation of more tasks, the request is routed by the hosting node to the master node. The nodes then create the actual tasks report back to the master, which in turn reports back to the initial node and task.

2.4 Graphical user interface library

This subsystem provides a basic graphical API wrapped around GTK+ and consists of widget and window classes that become the foundation for all GUI widgets in Scopira. More specialized and complex widgets, particularly useful to numerical computing and visualization, are also provided. This includes widgets useful for the display of matrices, 2D images, bar plots and line plots. Developers can use the basic GUI components provided to create more complex viewers for a particular application domain.

The Scopira graphical user interface subsystem provides useful user-interface tools (widgets) for the construction of graphical, scientific applications. These widgets complement the generic widgets provided by the GTK+ widget library with additional widgets for the visualization and inspection of numeric array data.

A matrix/spreadsheet like widget is able to view and edit data arrays (often, but not limited to matrices) of any size. This extensible widget is able to operate on Scopira *narray*s natively. The widget supports advances functionality such as bulk editing via an easy to use, stack based macro-language. This macro-language supports a variety of operations of setting, copying and filter selecting data within the array. A generic plotting widget allows the values of Scopira *narray*s to be plotted. The plotter supports a variety of plotting styles and criteria, and the user-interface allows for zooming, panning and other user customizations of the data plot. An image viewer allows fully zooming, panning and scaling of *narray*s, useful for the display of image data. The viewer supports arbitrary colour mapping, includes a legend display and supports a tiled view for displaying a collection of many images simultaneously. A simplified drawing canvas interface is included that permits software developers to quickly and easily build their own custom widgets. Finally, Scopira provides a *Lab* facility to rapidly prototype and implement algorithms that need casual graphical output. Users implement their algorithms as per usual, and a background thread handles the updating of the graphical subsystem and event loop.

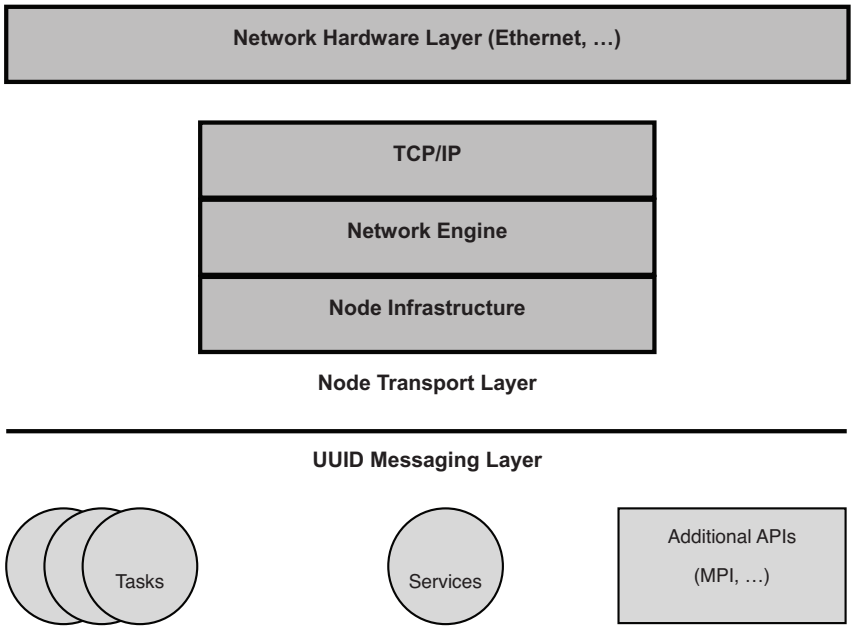

Fig. 3. The SAL network stack.

Scopira provides an architecture for logically separating models (data types) and views (graphical widgets that present or operate on that data) in the application. This view-model relationship is then registered at runtime. At runtime, Scopira pairs the compatible models and views for presentation to the user. A collection of utility classes for the easy registration of typical objects types such as data models and views are provided. This registration mechanism succeeds regardless of how the code was loaded; be it as part of the application, as a linked code library, or as an external plug-in.

Third parties can easily extend a Scopira application that utilizes models and views extensively. Third party developers need only register new views on the existing data models in an application, then load their plug-in alongside the application to immediately add new functionality to the application. The open source C++ image processing and registration library ITK (Ibáñez & Schroeder, 2005) has been successfully integrated into Scopira applications at run time using the registration subsystem.

A *model* is defined as an object that contains data and is able to be *monitored* by zero or more *views*. A *view* is an object that is able to bind to and listen to a model. Typically, views are graphical in nature, but in Scopira non-graphical views are also possible. A *project* is a specialized model that may contain a collection of models and organize them in a hierarchical fashion. Full graphical Scopira applications are typically project-oriented, allowing the user to easily work with many data models in a collective manner. A basic project-based application framework is provided for developers to quickly build GUI applications using *models* and *views*.

A complementary subsystem provides the base OpenGL-enabled widget class that utilizes the GTKGLExt library (Logan, 2001). The GTKGLExt library enables GTK+ based applications to utilize OpenGL for 2D and 3D visualization. Scopira developers can use this

system to build 3D visualization views and widgets, which allows for greater data exploration and processing. Integration with more complete visualization packages such as VTK (Schroeder et al., 2006) is also possible.

3. Agent based modelling framework

3.1 Agent based models

ABM simulations consists of emulating real-world systems in which collections of autonomous decision-making entities, distributed in a grid or lattice, interact with each other unveiling the dynamics of the system under investigation (Bonabeau, 2002; Sokolowski & Banks, 2010). This kind of simulation has emerged from the need to analyse and model systems that are becoming more complex in terms of their interdependencies, at the same time that data are becoming more organized into databases at finer levels of granularity. This type of modelling and simulation originated in the field of multi-agent systems, which derived from research in distributed artificial intelligence. The idea behind distributed artificial intelligence is to solve problems (or a related set of sub-problems) by distributing them amongst a number of relatively simple programs or agents, each with its own particular type of knowledge or expertise (Gilbert & Terna, 2000; Epstein, 2007). Additionally, ABM research draws from several related fields including computer science, management science, social science, system dynamics, complexity theory, complex networks, and traditional modelling and simulation (Macal & North, 2005; Shoham & Leyton-Brown, 2008).

The first step in defining an ABM is to establish the meaning of the term "agent". Although there is not one universally accepted definition, a good characterization may be given as:

"An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives." (Wooldridge, 1997). The most fundamental feature of an agent is the capability of making independent decisions and relating itself with other agents and their environment. From a practical modelling standpoint, an agent is a discrete individual, identifiable and located; goal-directed and autonomous; lives in an environment; has a set of intrinsic characteristics (properties), decision-making capabilities, a system of rules that govern its behaviour, and the ability to learn and adapt based on experience (which requires some kind of internal memory); can interact with other agents, and respond to the environment.

Depending on the subject under investigation, there may be different types of agents, for example trees, bugs, humans, viruses, bacteria, or cells. Even sub-types are possible, like tissue cells, immunological system cells, stem cells, or nerve cells. Considering the vast gamut of possible agents and situations, a system may be composed of more than one type of agent at the same time. For instance, assuming an infectious disease in a population of susceptible individuals, the variability of the species may be included considering that each agent may present different levels of susceptibility to the specific disease caused by differences in their immune system, lifestyle, prior health conditions, specific health risk factors, and so on.

The uniqueness of every agent is established by its state variables, or attributes, such as an identity number (desirably unique and immutable so that the agent can be traced during all the pertinent simulations), age, gender, location, size, weight, energy reserves, political opinion, disease stage, etc. These attributes distinguish an entity from others of the same kind, trace how the agent changes over time and guide its behaviour. In addition, these state

variables may be saved allowing re-starting the simulation once interrupted. It is from the heterogeneity of agents, and consequent variety of behaviours, that the collective aspects of interest concerning the real-world system being modelled and simulated emerge.

From the uniqueness and autonomous nature of the agents, it becomes quite clear that adopting an agent-based approach demands multiple agents in order to represent the decentralized perspective of the problem. Moreover, the agents must interact with one another and with the environment. For this purpose, the agents must be spatially located in some type of grid or lattice in which they can come into contact with other agents based on defined rules and assumptions.

The agent lattice awareness is limited and localized. Agents are always placed in some position of the grid and have limited visibility of their surroundings, implying that only a few other agents are in the agent's domain of influence. This is in contrast with a typical two-dimensional cellular automata, in which every position of the grid consists of a cell; each cell can assume different states that may be updated according to some rules based only on its immediate neighbours. In an ABM, there can be more than one agent per location of the lattice at the same time. The agents are not static, ensuring that they are free to move so their radius of interaction in the long term is not restricted to their nearest agent neighbours.

In contrast to many analytical models, which are in general easier to communicate and analyse since they are described using precise mathematical formulas, ABM descriptions are frequently incomplete and therefore less accessible to the reader. In an attempt to tackle this pertinency problem, a group of modellers in the field of ecology (Grimm et al., 2006) proposed a standard protocol for describing ABMs. The "Overview, Design Concept and Details" protocol, or simply ODD, proposes that an ABM (and its various interaction rules) be described in a standardized way making model descriptions easier to understand and complete. Achieving this objective makes ABMs reproducible and a much more reliable scientific tool for investigations.

3.2 Designing an agent based model

Before modelling a problem using an agent-based approach, a few points must be considered. First, one must consider the nature of the problems that can benefit from this method and how the simulations can provide useful information about the scenarios being investigated. An ABM can provide a realistic and flexible description of a system, and it can capture emergent phenomena (Bonabeau, 2002; Shoham & Leyton-Brown, 2008). An "emergent phenomenon" can be conceptualized as a large scale, group behaviour of a system, which does not seem to have any clear explanation in terms of the systems constituent parts (Darley, 1994; Sokolowski & Banks, 2010). These phenomena may arise due to the nonlinear nature of the behaviour of the agents, which can be described by discontinuities in agent behaviour that is difficult to capture with differential equations. In addition, agent exhibits memory (non-markovian process), path-dependence, learning and adaptation. Being so, ABMs are adequate for describing real world phenomena where agent behaviour is complex (Ferguson, 2007). Upon concluding that an ABM is appropriate for a specific case, the chart in Figure 4 offers some guidelines for the development of an ABM.

Modelling agent-based simulations requires the creation of a representation of the "sub-world" under investigation. For this representation to be realistic and accurate, there should be a set of available **data** to support the various considerations adopted for the several aspects and parts included in the agent-based model. As we go through the design process

of an ABM for the spread of an infectious disease in an urban centre, we are going to illustrate what data can be used in each step.

Considering the spread of an infectious disease in a population, data regarding the age-gender structure of the population can be obtained and used to create the *in silico* population. However, the lack of detailed information regarding the actual event (e.g., estimates of epidemiological parameters for an emerging infectious disease) often poses significant challenges for modelling agent-based simulations. On the other hand, experimenting with the model under various assumptions (for instance, different intervention strategies) may reveal emergent phenomena that lead to a better understanding of the system and its behaviour. It is this circular dependency relationship that makes ABMs powerful and challenging at the same time.

Although this modelling framework was built to be generic for major types of ABMs, it is possible that some aspects of very specific cases have not been included. Nevertheless, with this framework, the reader should have a starting point for designing an ABM and performing simulations by adding specific characteristics of the system. As with any software development project, the main point to be defined is the **purpose** of the application. In our example, the purpose of the model is to understand what are the key characteristics of a population, with its social interaction patterns, that influences the spread of an infectious disease and leads to epidemic scenarios. In this stage, the appropriate **disease model** for the specific disease must be chosen. Unless the disease under investigation is completely new and uncharted, it is possible to choose whether a simple susceptible-infectious-recovered (SIR) model suffices or if a more refined one, including other intermediate states, is required. It is important to note that the assumptions based on **global community parameters** influence some of the **local individual parameters**. In this example, choosing an SIR model implies that, in terms of the disease, the agents can be in only one of the three (susceptible, infective, or recovered) states. From the perspective of the local individual parameters, aspects such as pregnancy, aspirin treatment, and so on, can be taken into account.

This description gives the general idea of what is going to be simulated. At this point it is important to know the exact questions the model is aimed to answer. This way it will be clearer what data resulting from the simulation should be stored for analysis. Once the purpose is established, we will have a clear idea of what the agents in our simulation are going to be. In general, simulations will involve different kinds of agents. In a typical spread of some virus in a population, as with the 2009 H1N1 influenza pandemic, there is only one type of agent involved, namely humans. These agents are either affected by the disease or not. When infected, they can transmit the disease to a susceptible agent if they come to a close contact range (the definition of close contact is all interactions that can result in infection). In this case, there is no need to have more than one agent since the infectious disease may be considered as a property of the agent. Conversely, if the interest is to investigate the pattern of transmission of a disease from pigs to humans, then two types of agents may be involved (pigs and humans). In more specific scenarios, the interest may be in the virus population itself. In this case, there can be sub-types of agents, i.e. virus strains, which are all sub-types of a broader class of a specific virus.

Regardless of the type of agents, they will have features (state variables) that define them as individuals. For all agents, the modeller will need to list all the state variables including not only those that define their characteristics (like age, weight, and temperature), but also those that describe the agent's behavioural traits, such as velocities, strategies, probabilities, susceptibility, etc. In addition, it is important to define how the agents interact with the

environment within specified time periods. Demographic data is used to create the agents so that the resulting *in silico* population follows the same distributions as the real population. In this sense, the state variables are defined as the primitive characteristics of the individuals that can be later described in terms of demographic information, such as age and gender profiles. Additionally, several other, non-demographic specific, variables must be defined. Some of these variables relate to the disease, such as symptoms duration, susceptibility, etc. Other kinds of variables relate to behavioural traits. Based on these variables and relevant information acquired from the environment and other agents, an agent can have its state altered and make decisions regarding its next steps. In short, the state variables are the most important part of the model, given that they define the agents and provide the relevant information for them to act in an autonomous fashion.

Once agents are defined, the next step is to determine where they are placed, thus defining their **environment**. If the relationship between the agent and the environment is not important for simulation purposes, the environment should be only a place that facilitates the interactions between agents. In some cases, this relationship may be important, as in the case when the demographic distribution may play an important role in the spread of an infectious disease. Once the environment is defined, its scales must be determined to establish its dimension and the size of each dimension. Additionally, the representation of the environment is not restricted to an n-dimensional space but rather defined as a complex network or some other data structure, and may contain other properties that influence the agents' decisions and behaviour. In epidemic scenarios, urban centre characteristics such as commuting distance, demographic density, and city boundaries, may affect the spread of the infectious disease, as well as agent behaviour. Being so, data regarding mobility patterns can be used to define lattice properties, which in turn can be used by the agents in its decisions and actions.

The planning of the agents and environment is not complete until the **processes** associated with them are defined, which leads us to the next block in the flowchart in Figure 4. The set of processes are the kernel of the simulation. At this stage, it should be pointed out that all actions that each agent takes are in terms of updating its state, interacting with other agents, and interacting with the environment. Likewise, all the mechanisms for updating the environment should be listed.

In terms of updating the state of an agent, several steps may be considered, such as deciding if the agent's objective has been achieved, or identifying the best course of action for achieving the objective. As an example, considering the case of an infectious disease, it must be verified if the agent is infected and, if so, to decide if it is time to seek medical care. Performing agent-to-agent (or agent-to-environment) interactions requires communication between the agents (or with the environment), so that the entities involved can access each other's information and choose their next action. In the case of a susceptible agent interacting with an infective agent, it must be determined whether the susceptible individual will become infected.

Once all the associated processes are defined, a natural question that arises is the timing at which these actions take place. This issue is represented by the **scheduling** box in Figure 4. Here, a flowchart of the processes will be useful for determining the order that the processes are performed. Several processes may occur concurrently, while others may depend on other processes. Knowing the overall picture of the events, it is possible to identify if the state variables are updated synchronously or asynchronously. Moreover, the type of time-scheme should be determined in terms of being discrete, continuous, or both.

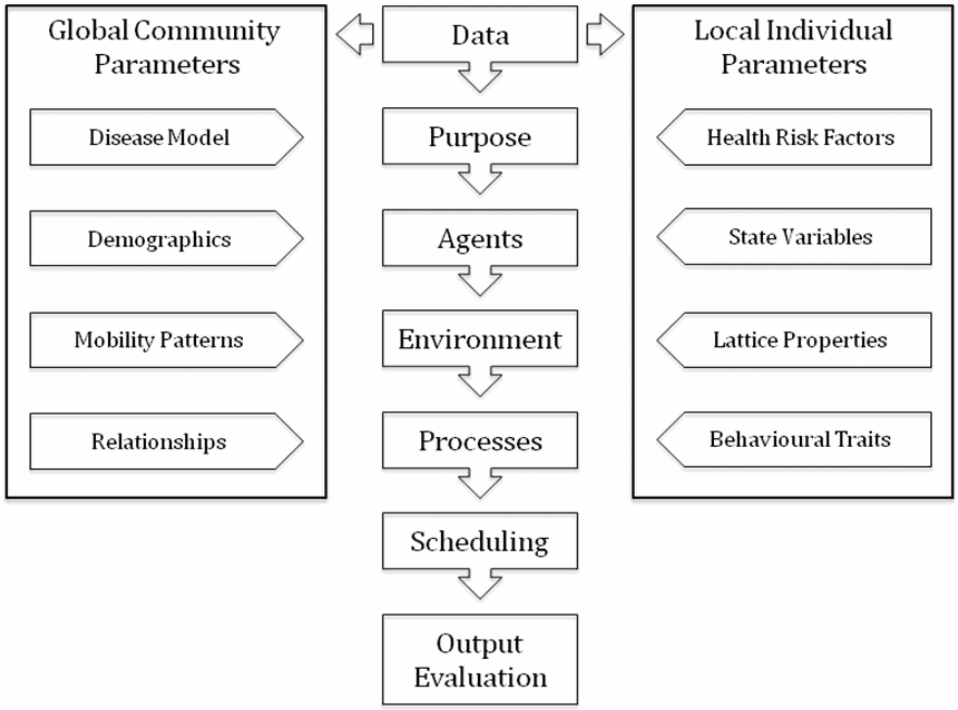

Fig. 4. Development cycle for an agent-based infectious disease model.

In the context of parallel computing, the relationship among processes will determine the ease with which to parallelize the code (for example, tightly coupled or spatially proximal relationships are more difficult to parallelize compared to loosely coupled or spatially distant relationships). ABMs may start with small and simple simulations. However, as the ABM evolves over time, it can become large and computationally complex, thus requiring the parallelization of the execution of the code. In this way, structuring the model before coding, and choosing the right tools for the task, will certainly save time in the end.

3.3 Building an agent-based simulation

In this section, we describe an ABM software framework for simulating the spread of a contagious disease in human populations. The agents in this model are a simplified representation of the real inhabitants of an urban centre in which an infectious pathogen was introduced. In this system, the three main components of simulations are: the agents, the lattice (or grid) in which these agents are placed, and the rules of between-agent and agents-environment interactions. Considering that ABM simulations rely on the interactions between agents that are similar in characteristics but have different behaviours, it is natural to choose any object oriented programming (OOP) language to run such simulations. Using OOP, each agent may be created as a representation of a common class of agents in which every software object has the same set of properties, or state variables. Assuming a different set of values for the properties of each agent, the uniqueness of each agent is established, and the variability of the species is taken into account (Grimm et al., 2006; Epstein, 2007).

In the context of an ABM, the unit of simulations is the agent (often referred to as an individual), and in the context of OOP, this unit is the object of a defined class. It is, however, important to note that agents and software objects are different components of the ABM. A software object is an appropriate way to represent agents in the sense that objects encapsulate both agent state (as data members) and agent behaviour (as methods), and communicate via message passing. But agents, on the other hand, are "rational decision making systems" that are able to react and behave according to the situation (Wooldridge, 1997).

Depending on the particular problem being modelled, the agents' properties of interest may vary. For an epidemic spread in a population of susceptible agents, some of the relevant characteristics of these agents include age, gender, susceptibility to the disease, state of the disease, among several other properties pertinent to the nature of disease in infectious agents. In such a scenario, investigating the dynamics and evolution of the contagious disease requires the development of a realistic model of the population with a significantly large number of agents (as given in pertinent demographic data) whose properties render the age-gender distribution of that specific location. Hence, the performance of the simulation framework becomes an important issue when choosing the proper tools for building the simulation.

As with every software engineering project, choosing the appropriate programming language and libraries to build the ABM application is crucial to produce reliable, efficient, and adaptable software code as described in Section 1. In an ABM, the use of OOP is an appropriate simplifying approach for the logic of simulations and coding processes. Furthermore, the choice of the right model data and libraries will impact the performance and resource utilization, reflecting in the last instance in the running time of the simulation. Keeping in mind these requirements, we adopted C++ as the most suitable programming language for the ABM described here. In terms of performance and versatility, the C++ language is comparable to the C language with the addition of some OOP specializations. Moreover, using the Scopira framework (see Section 2) functionalities and data structures like smart pointers, *narray*s and its distributed computing tools, simplifies the software development of ABM applications especially with respect to code parallelization.

Another important characteristic of ABMs is related to the stochasticity present in every part of the simulation. The variability among agents may arise from sampling, for example, an age-gender distribution, a susceptibility distribution, or distribution of infectious period. The state of an agent can be altered by means of interactions with other agents, where the outcome of these interactions depends on the value of parameters sampled from some prescribed distribution. As a result, the random number generators must be chosen carefully. For the generation of random numbers, a good choice of numerical library is the *GNU Scientific Library*, which has been widely adopted in scientific and engineering applications and which has the additional advantage of being open source, thread safe, well tested, and portable (Gough, 2009).

3.4 Simulation framework design

Agents are modelled as objects of a class named *Agents* in which the member variables represent the characteristics of the agents, such as age, gender, susceptibility to the disease, and state of the disease. The methods of this class are simple manipulators that change the values of these member variables. In this way, every agent represents a person with the same basic characteristics that may assume different values. Using age-gender distributions for a specific population, variability in age and gender is introduced. In an epidemic disease

scenario, an agent may assume different states regarding its epidemiological status, which may be one of the following: susceptible, exposed (but not yet infectious), infectious and recovered (immune against re-infection). Since the number of agents in a simulation may range from a few thousands to millions, it is important to save memory whenever possible. Considering that these states of infectious may assume only two values (*true* or *false*), stating whether the agent is in the specific state or not, the best way to store this information is to use one *unsigned int* as a bit set (or alternatively to use the *bitset* class from the C++ standard library), in which every bit represents one of the possible states. Furthermore, the state of the agent may be assessed by using bitwise operations testing only the specific bits of interest, saving in this way, a few processor operations per agent per cycle.

Similar to a real world scenario, agents, in general, live in a city (or community) in which they must be located somewhere and are free to move to different locations. Therefore, the city is modelled as another class, named *Lattice*, in which a 2-dimensional n-array is used to represent the city map and is a container to hold agents. The simplest way to build ABM simulations is to consider only one agent per site that interacts with its adjacent neighbours, as in a cellular automaton, or even with distant neighbours, but carrying these interactions without moving from its position. Another approach, which is used in the model presented here, is to consider several free-to-move agents at the same position. This model tries to mimic the behaviour of people in a city, where it is possible for people to interact with others in the same location, as would be the case in a house, office, or a shopping mall.

Considering that several agents may be in the same *x*-*y* position of the city grid at the same time, and that these agents may move through the map, every site in the lattice holds a standard vector of Scopira's smart pointers to agents, that is, the lattice is a two-dimensional n-array of standard vectors of pointers to agents as shown in the left panel in Figure 5. Based on this scheme, moving an agent from one location to another is done by assigning a pointer to the agent being moved to the vector at the destination location, and then removing the pointer to this agent from the vector at the initial position, as shown in the right panel in Figure 5.

Using this smart pointer strategy, there would be no additional overhead of moving or copying the agent's data into memory. Another advantage with smart pointers is that there is no need to manually free the memory at the end of the application, since it is refreshed as soon as the smart pointer goes out of the scope. Furthermore, it is possible to test if the software object is alive, which dramatically reduces the number of segmentation faults during the software development phase due to access attempts to memory pages that have already been released. Additionally, by using scope as a way to control the life cycle of the software objects, the resulting application is better structured, robust and extensible.

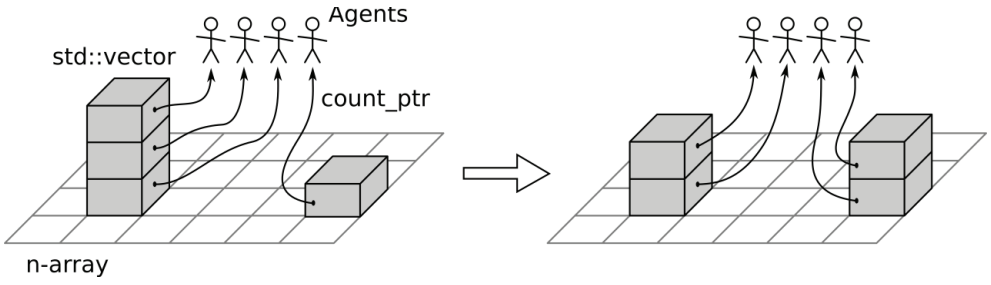

Fig. 5. Representation of agent movements in the lattice.

The methods of the *Lattice* class are used to control the access to the map locations and to the agents located in each position. Using, for example, the map of a city in which its boundaries and demographic distribution is specified, the lattice representation will define all the sites (that is, *x-y* positions) according to the city boundaries, and the number of agents allowed in each site according to the demographic density. In this approach, the agents are placed in the lattice but they may not be directly related. The agent-to-agent and agent-to-lattice relationships are stored in another class called *Simulation*. In this class, a smart pointer to the lattice is used, and the rules concerning the movement of the agents in the lattice and interaction between agents are considered as methods of this class. Furthermore, this class is responsible for updating counters that keep track of the number of agents in each state in the simulation, and for writing reports to files on the computer system's hard disk drive. For these two tasks, the *Counters* and *Reports* classes are generated.

Since the simulation has been built to be able to use more than one processor using threaded programming, *Counters* and *Reports* classes need to be built in order to be thread safe, thereby supporting concurrency. When working with multiple threads, it is possible that different threads will try to access the same memory area simultaneously; for example, to increment the value of the same variable at the same time; this may cause unpredictable results. One way to avoid data inconsistencies is to use mutual exclusion locks, or mutexes. Before a thread operates with a variable, it must acquire the lock on this variable in order to prevent other threads from operating on the same variable at the same time. This ensures serialization of access to the specific data area. After the thread has performed the desired operations, it needs to unlock this variable for other threads to access it. Determining what resources should be locked and unlocked, and in what order, are fundamental considerations when parallel programming. Moreover, a strategy must be adopted for interacting with these shared resources in order to minimize the number of simultaneous accesses to the same shared area. Otherwise, it will slow down the simulation considering that when a thread is waiting to acquire the lock, it stops its other operations. Although it is possible to tackle this situation in different ways, it makes the problem more complex and might introduce other unnecessary synchronization issues.

The *Counters* and *Reports* classes both perform actions on areas of common access by threads. The member variables of the *Counters* class, keep track of several aspects of the simulation, like the current time step in which the simulation is run, the total number of susceptible agents, and the number of infectious agents. Scopira provides several methods for parallel programming in SAL that simplifies this process. In the case of protecting certain data areas like variables, it defines a *shared_area* class to which is associated a mutex that can be later accessed by objects of the *lock_ptr* class. Auxiliary to the *Counters* class, it defines a data structure called *SimulationCounters* that holds all the counter variables that may be accessed by more than one thread at the same time. In the *Counters* class, an instance of the *shared_area* class associated with the *SimulationCounters* data structure is created. The methods of the *Counters* class, when manipulating these variables, call them through an object of the *lock_ptr* class associated with the object of the shared area. When the lock pointer is created, it automatically locks the shared area with which it is associated, and when it moves out of scope, it is automatically destroyed, thereby unlocking the associated shared area. Likewise, when the method is called, it creates the lock pointer and acquires the lock, then the desired operations are performed, and at the end of execution of the method, the lock pointer is destroyed automatically, hence unlocking the shared area. With this method, the programmer does not need to manually lock and unlock the variables, but rather needs to define the lock pointer inside the scope targeted for manipulation of the shared area.

Following the same idea, the *Reports* class is used to write information about the simulation into files. If two threads try to access the same file at the same time, the message may become garbled, and therefore mutexes must be used. For this purpose, Scopira provides the *fileflow* class that in turn provides methods for manipulating files and also includes mutexes. An object instance of the *Reports* class is associated with each file. It is basically responsible for opening and closing files, and overloads the stream insertion operator "<<". In this way, every time the insertion operator is called on a *Reports* object, it calls the *fileflow::write_lock()* method, passes the message to the *fileflow* object, and then unlocks it through the *fileflow::unlock()* method.

Figure 6 shows a diagram of the simulation environment with its several components. In this representation, the *Simulation* class aggregates *Lattice*, *Reports*, and *Counters* classes. These three classes are included in the *Simulation* class via smart pointers to the appropriate objects of each class. Since the *Lattice* class contains smart pointers to all the agents in the simulation that can be accessed through the *Lattice* object, the rules concerning the agent-to-agent and agent-to-lattice interactions are methods of the *Simulation* class.

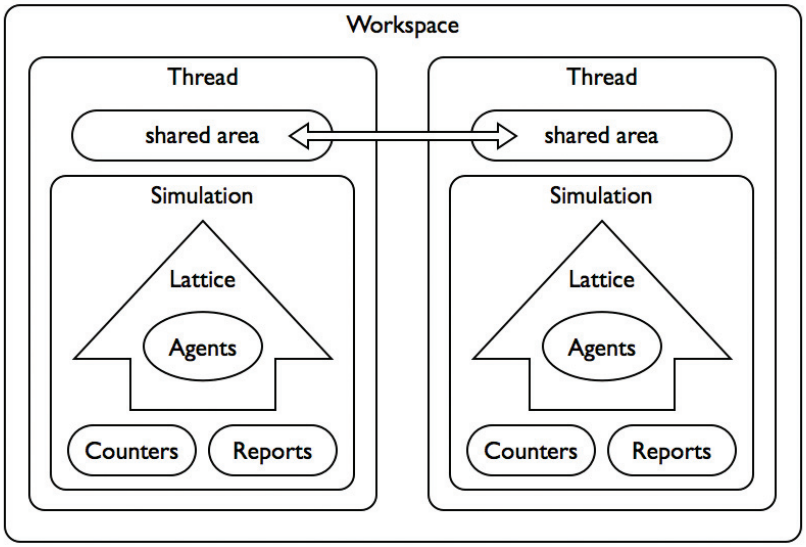

Fig. 6. Simulation environment showing the interaction of two threads.

For the agent-based simulation infrastructure proposed here, we now present the mechanisms for parallelizing the simulations. The parallelism strategy adopted in this example was to divide the lattice into chunks and assign them to different processing threads. To balance the amount of work performed by each thread, the *narray* that represents the lattice was divided into an approximately equal number of active sites, which correspond to the defined locations in the city map. Agents are allowed to move between sites in the map but the location of the destination site may be out of the scope of control for the actual processing thread. As a result, it is necessary for all threads to be equipped with a shared area to which other threads can send the moving agents (see Figure 6). The *Threads* class inherits the basic functionality of the *scopira::thread* class, which provides the mechanisms for creating the parallel environment and controlling all aspects of the threads

independent of the operational system. As a result, only a few methods concerning the particular problem being investigated and a shared area for exchanging agents between threads need to be implemented in the *Threads* class. The most important method, from the simulation point of view, is the *Threads::run()* method, which is an overload of the Scopira's *thread::run()* method, where the actual simulation takes places. The pseudo-code for this method is given below.

```
Threads::run() {
        CheckSharedArea()

        foreach site:
                simulation.InteractAgentsToAgents( site )
                simulation.InteractAgentsWithLattice( site )
                simulation.MoveAgents( site )

        CheckSharedArea()
        }
```

In this case, every thread iterates over all the sites under its control, and performs the tasks required by the simulation. In a simple case, it will perform the agent-to-agent and agent-to-lattice interactions, and move the agents if necessary. The latter step of moving agents is the most critical task in terms of distributed programming, as it involves other threads and synchronization issues.

As mentioned before, an agent may move to a location out of the scope of control for its current assigned thread. For this reason, the controlling thread must verify if the agent is moving out of its domains and send it to the shared area of the responsible thread. Consequently, additional steps in the *run()* method are required, including verification of its own shared area, which is performed by the *CheckSharedArea()* method, and attribution of the moving agent to its new location, performed by the *MoveAgents()* method. At this point, synchronization issues emerge requiring strict control over the time step of the agents.

Although threads share the same address space, they are not aware of other threads and should be controlled from a central location, namely the parent process. The *Workspace* class is responsible for creating, control, destroying, and keeping track of the threads and their domains, as well as facilitating their communication. In order to perform these activities, each thread must have a reference to the instance of the *Workspace* that can be used as a mediator between all active threads.

4. Concluding remarks and outlook

The current spectacular interest in agent-based modelling has gradually built up over the last twenty years, in particular for understanding the social aspects of human populations and simulating the spread of infectious diseases within and between communities. The use of agent-based models, in general, requires a more comprehensive incorporation of agents' characteristics both individually and group-wise, detailed information of the pertinent environment and the relationship between the system's various components. Despite the rapid evolution of ABM-based software applications and development of more sophisticated simulation approaches, the study of ABMs of any kind lacks a comprehensive and flexible software development framework. While some efforts have been made on developing such simulation models more consistent with the nature of the systems under

investigation, and on designing computer software algorithms for their rapid implementation, the literature on general theoretical aspects of agent-based models is, as yet, quite small.

In this chapter, we reported our attempts in developing a software simulation framework for agent-based infectious disease modelling. Models implemented using this development framework will satisfy the software objectives of reliability, efficiency, and adaptability as fully described in Section 1. This framework has already been used to implement an agent-based model to evaluate mitigation strategies applied during the 2009 influenza pandemic in urban centres. An important aspect of this development framework is its flexibility to be adopted for simulating interconnected populations with distinctly different mobility patterns and demographic structures. Strategies for implementing this framework to simulate the spread of a disease in remote and isolated populations are being currently investigated.

The agent-based modelling framework described has several advantages that go beyond computer simulation experiments, providing a platform for addressing important aspects of modern world with global connectivity. We plan to use this framework to develop desktop decision-support systems for use in public health to address critical issues arising in the acute management of public health crises. Incorporating agent-based models into these software systems can provide an essential tool for public health experts to perform preliminary analysis, which can inform the formulation of optimal mitigation strategies in the face of substantial uncertainty regarding epidemiological aspects of a novel disease. Such decision support systems will require rapid development and deployment, an intuitive graphical user interface, and must quickly produce scenario outcomes. Such requirements can be satisfied using the framework presented here.

In the context of software design and engineering, future research activities will include the development of fault-tolerant distributed agent-based modelling systems as well as a comprehensive model description markup language to generate efficient software. The end results of these activities must satisfy the requirements of both modellers and public health officials in simulating the outcomes of infectious disease transmission, as well as prevention and control strategies.

5. Acknowledgment

The research activities and software engineering described in this chapter were supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Mathematics of Information Technology and Complex Systems (MITACS), Canadian Institutes of Health Research (CIHR), and the Department of Foreign Affairs and International Trade of Canada, Emerging Leaders of the Americas Program.

6. References

Bernoulli, D (1760). Essai d'une nouvelle analyse de la mortalité causée par la petite verole, *Histoire de l'Académie Royale des Sciences, Memoires, Année*, 1–45

Bonabeau, B. (2002). Agent-based modeling: methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, Vol. 99, Supplement 3, 7280–7287, ISSN: 1091-6490

Darley, V. (1994). Emergent phenomena and complexity, In: *Artificial life IV, Proceedings of the 4th International Workshop on the Synthesis and Simulation of Living Systems*, Brooks, R. & Maes, P. (Ed.), 411–416, MIT Press, ISBN: 0262521903, Cambridge

Demko, A. B. & Pizzi, N. J. (2009). Scopira: an open source C++ framework for biomedical data analysis applications. *Software: Practice and Experience*, Vol. 39, No. 6, 641–660, ISSN: 1097-024X

Dongarra, J. J. & Dunigan, T. (1997). Message-passing performance of various computers. *Concurrency and Computation: Practice & Experience*, Vol. 9, No. 10, 915–926, ISSN: 1532-0634

Epstein, J. M. (2007). *Generative Social Science: Studies in Agent-Based Computational Modeling*, Princeton University Press, ISBN: 0691125473, Princeton

Ferguson, N. M. (2007). Connections capturing human behaviour, *Nature*, Vol. 446, 733, ISSN: 0028-0836

Geddes, K.; Labahn, G. & Monagan, M (2008). *Maple 12 Advanced Programming Guide*, Maplesoft, ISBN: 9781897310472, Waterloo

Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R. & Sunderam, V. S. (1994). *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, ISBN:0262571080, Cambridge

Gilbert, N. & Terna, P. (2000). How to build and use agent-based models in social science. *Mind & Society*, Vol. 1, No. 1, 57–72, ISSN: 1593-7879

Gough, B. (2009). *GNU Scientific Library Reference Manual*, Network Theory Ltd., ISBN: 0954612078

Grimm, V.; Berger, U.; Bastiansen, F.; Eliassen, S.; Ginot, V.; Giske, J.; Goss-Custard, J.; Grand, T.; Heinz, S. K. & Huse, G. (2006). A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, Vol. 198, No. 1–2, 115–126, ISSN: 0304-3800

Grimm, V. & Railsback, S. F. (2005). *Individual-based Modeling and Ecology*, Princeton University Press, ISBN: 069109666X, Princeton

Hill, F. S. &, Kelley, S. M. (2006). *Computer Graphics Using OpenGL (3rd Edition)*, Prentice Hall, ISBN: 0131496700, Upper Saddle River

Ibáñez, L. & Schroeder, W. (2005). *The ITK Software Guide: The Insight, Segmentation and Registration Toolkit*, Kitware, Inc., ISBN: 1930934157, Clifton Park

Kermack, W. O. & McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics, *Proceedings of the Royal Society London A*, Vol. 115, 700–721, ISSN: 1471-2946

Kitware, Inc. (2010). *VTK User's Guide: Install, Use and Extend the Visualization Toolkit*, Kitware, Inc., ISBN: 1930934238, Clifton Park

Krause, A. (2007). *Foundations of GTK+ Development*, Springer-Verlag, ISBN: 1590597931, New York

Lingappa, J. R.; McDonald, L. C.; Simone, P. & Parashar, U.D. (2004). Wresting SARS from uncertainty. *Emerging Infectious Diseases*, Vol. 10, No. 2, 167–170, ISSN: 1080-6059

Logan, S. (2001). *GTK+ Programming in C*, Prentice Hall, ISBN: 0130142646, Upper Saddle River

Macal, C. M. & North, M. J. (2005). Tutorial on agent-based modeling and simulation, *Proceedings of the 37th Winter Simulation Conference*, pp. 2–15, ISBN: 0780395190, Orlando, USA, December 4–7, Winter Simulation Conference, Orlando

Moghadas, S. M.; Pizzi, N. J.; Wu, J. & Yan, P (2009). Managing public health crises: the role of models in pandemic preparedness. *Influenza and Other Respiratory Viruses*, Vol. 3, No. 2, 75–79, ISSN: 1750-2659

Murray, J. D. (2007). *Mathematical Biology: Vol. I. An Introduction (3rd Edition)*, Springer-Verlag, ISBN: 0387952233, Heidelberg

Ormerod, P. & Rosewell, B. (2009). Validation and Verification of Agent-Based Models in the social sciences, In: *Epistemological Aspects of Computer Simulation in the Social Sciences*, Squazzoni, F. (Ed.), 130–140, Springer, ISBN: 364201108X, Berlin

Pizzi, N.J. (2008). Software quality prediction using fuzzy integration: a case study. *Soft Computing Journal*, Vol. 12, No. 1, 67–76, ISSN: 1432-7643

Pizzi, N. J. & Pedrycz, W. (2008). Effective classification using feature selection and fuzzy integration. *Fuzzy Sets and Systems*, Vol. 159, No. 21, 2859–2872, ISSN: 0165-0114

Railsback S. F.; Lytinen, S. L. & Jackson, S.K. (2006). Agent-based simulation platforms: review and development recommendations, *Simulation*, Vol. 82, No. 9, 609–623, ISSN: 0037-5497

Ropella, G. E. P.; Railsback, S. F. & Jackson, S. K. (2002). Software engineering considerations for individual-based models. *Natural Resource Modelling*, Vol. 15, No. 1, 5–22, ISSN: 1939-7445

Schroeder, W.; Martin, K. & Lorensen, B. (2006). *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, ISBN: 0139546944, Upper Saddle River

Shan, H.; Singh, J. P.; Oliker, L. & Biswas, R. (2003). Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, Vol. 29, No. 2, 167–186, ISSN: 0167-8191

Shoham, Y. & Leyton-Brown, K. (2008). Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations, Cambridge University Press, ISBN: 0521899435, Cambridge

Sigmon K. & Davis, T.A. (2010). *Matlab Primer (8th Edition)*, Chapman & Hall/CRC Press, Inc., ISBN: 1439828628, Boca Raton

Snir, M. & Gropp, W. (1998). *MPI: The Complete Reference (Volume 1: The MPI Core)*, MIT Press, ISBN: 0262692163, Cambridge

Sokolowski, J. A. & Banks, C. M. (2010). *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*, John Wiley & Sons, Inc., ISBN: 0470486740, Hoboken

Summerfield, M. (2010). *Advanced Qt Programming: Creating Great Software with C++ and Qt 4*, Prentice Hall, ISBN: 0321635906, Upper Saddle River.

Taubenberger, J. K. & Morens, D.M. (2006). 1918 influenza: The mother of all pandemics. *Emerging Infectious Diseases*, Vol. 12,No. 1, 15–22, ISSN: 1080-6059

Wolfram S. (1999). *The Mathematica Book*, Cambridge University Press, ISBN: 0521643147, Cambridge

Wooldridge, M. (1997). Agent-based software engineering. *IEE Proceedings: Software Engineering*, Vol. 144, No. 1, 26–37, ISSN: 1364-5080