Science & Technology
Facilities Council

Technical Report
RAL-TR-2012-019

# FLAME-II: a redesign of the Flexible Large-scale Agent-based Modelling Environment

LS Chin, DJ Worth, C Greenough, S Coakley,
M Holcombe, M Gheorghe

November 2012

**ISSN 1358- 6254**

# FLAME-II: A redesign of the Flexible Large-scale Agent-based Modelling Environment

L.S. Chin[†], D.J. Worth[†], C. Greenough[†]
S. Coakley[‡], M. Holcombe[‡] and M. Gheorghe[‡]

November 16, 2012

## Abstract

This report reviews the current design of FLAME and highlights its limitations. This is followed by the description of a proposed re-design which will overcome the current limitations and enable the exploration of various optimisation and parallelisation strategies. Some of these strategies are briefly discussed, along with other concerns such as backward compatibility and agent topology management.

† Software Engineering Group, STFC Rutherford Appleton Laboratory
‡ Computer Science Department, University of Sheffield

**Keywords:** agent-based modelling, parallelisation, optimisation, flame framework

Email: {`shawn.chin, christopher.greenough, david.worth`}`@stfc.ac.uk`
{`s.coakley, m.holcombe, m.gheorghe`}`@sheffield.ac.uk`

Reports can be obtained from: `http://epubs.stfc.ac.uk`

Software Engineering Group
Scientific Computing Department
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
Oxfordshire OX11 0QX

## Contents

## 1 Introduction

Over the years, FLAME [1] has evolved based on the requirements of different projects starting from a position-aware framework used for biological agent modelling [2][3][4] to a position-agnostic framework driven by a static scheduler and message board library catering to economic models [5].

Some of the functionality provided by FLAME was introduced as ad hoc features for particular models, while many of the architectural design and data structures used within the framework are a direct result of iterative improvements to meet project-specific goals. For the models it was developed for, FLAME was fit for purpose. However, as a generic framework for agent-based modelling there is still much room for improvement.

In this report, we briefly describe the current design and its limitations, followed by a discussion on the plans to re-engineer the inner workings of the framework from the ground up. The key considerations for this re-engineering effort are performance – to extract more parallelism from the simulation – and extensibility – to support different execution back ends, parallelism paradigms, and modelling domains.

## 2 Overview of the design of FLAME-I

In FLAME, modellers define agents using the concept of Communicating Stream X-Machine [6] (CSXM); each agent is represented by an acyclic state machine that characterises the behaviour of the agent per iteration.

Each state transition function has access to the internal memory of the agent, as well as input and output streams of information. In FLAME the input/output streams take the form of message boards [7]. Since message boards are the only means in which the agent communicates with the environment and other agents, this makes the agent model inherently parallel. Each agent can be executed independently as long as the input message board contains the expected messages.

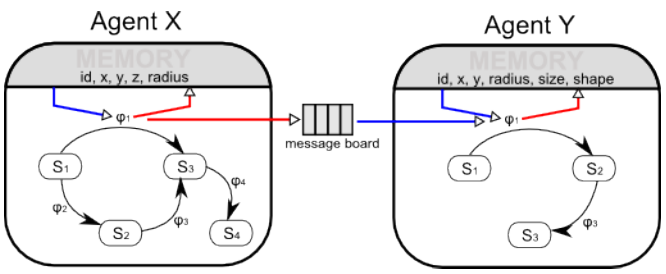


Figure 1: Agents as Communicating Stream X-Machines

The simulation can therefore be parallelised by distributing agents across disparate processing nodes and synchronising the message boards to ensure that all agents see the same set of messages.

For efficiency, agents are not allowed to read and write to the same board from the same transition function. This avoids the need to synchronise the boards on every single write operation.

The synchronisation of a board is initiated the moment all writes have completed using the `MB_SyncStart` function. This function is non-blocking and the synchronisation process is performed in a background thread. The framework is then free to execute other functions that do not depend on the board in question. It is possible for multiple message boards to be synchronised concurrently.

Before executing agent functions that reads messages from a board, the `MB_SyncComplete` function has to be called. This function checks the status of the synchronisation process and

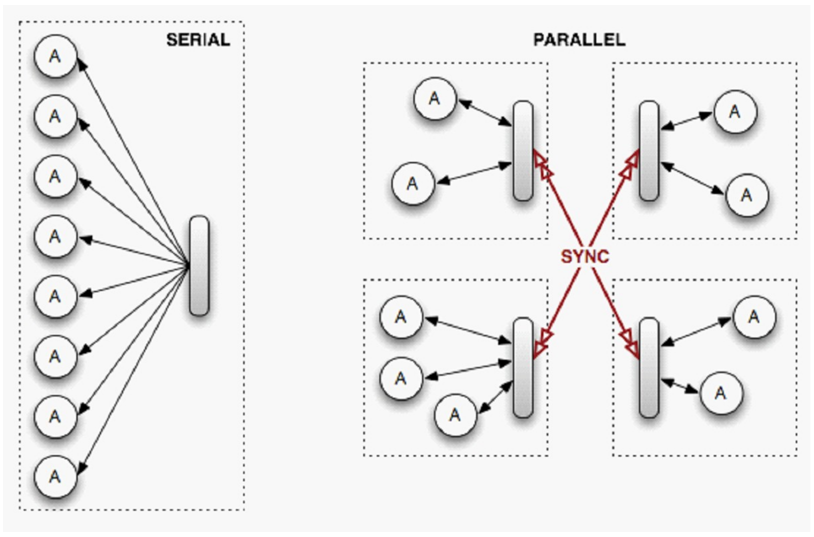


Figure 2: Parallelism achieved by distributing agents and synchronising message boards

returns immediately if the synchronisation is complete. However, if synchronisation is still in progress the function blocks until completion.

An important aspect of attaining good performance is therefore to completely hide the communication cost by scheduling as much computation as possible between functions that write messages and those that read them.

At present, the scheduling of functions is done statically. This is done by the *xparser* — the FLAME model parser which parses the agent definition (written in a dialect of XML called XMML) and generates the simulation code.

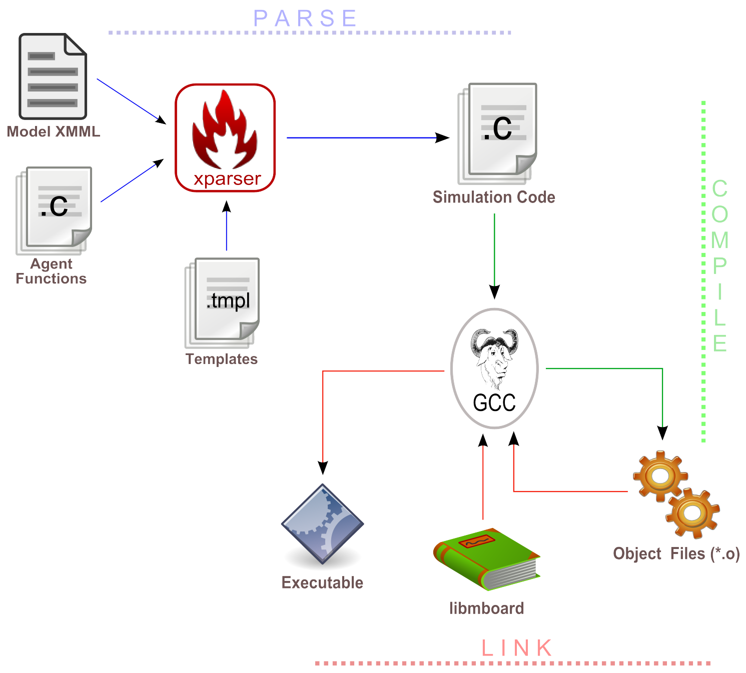


Figure 3: Building a FLAME simulation

Based on the model definition, the *xparser* produces a directed acyclic graph representing a dependency graph of transition functions. Each agent model will have its own function dependency graph and they are coupled together by dependencies on message boards. Nodes representing a message boards are dependents of functions that write to the board, and dependencies of functions that read from the board.

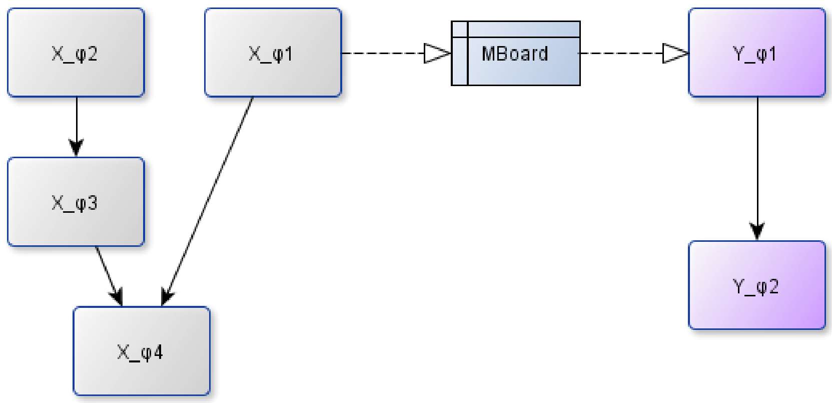


Figure 4: Model represented as function dependency graphs coupled together by dependency on message board

Using the function dependency graph, the *xparser* can schedule the execution of agent transition functions such that message producers are scheduled as early as possible and message consumers as late as possible. This maximises the amount of computation being performed while the synchronisation process is in flight.

It is possible for transition functions of different agent types to be interleaved as long as the dependencies are met.

Agent instances are represented as a `struct` containing the internal memory of the agent. Agent transition functions read this memory `struct` and updates its values, effectively transitioning the agent instance to the next state ready to be consumed by the next function. The execution of a transition function is repeated for all agent instances of the associated type in the relevant state. Once all functions have been called (in the correct order so as to meet dependencies) an iteration of the simulation is complete.

## 3 Limitations of the Current Design

To define an agent, modellers specify a set of state transition functions to transition an agent from one state to another. When linked together, these transition functions and their associated states form the acyclic state machine that represents the behaviour of the agent.

These functions have read-write access to all variables within agent memory which seems sensible at first, but in hindsight is the cause of (or a contributing factor to) some of the limitations of the design.

### 3.1 Data Granularity

Because each function can potentially write to all memory variables, the smallest unit of data is the whole agent instance. Data partitioning for parallel execution has to therefore be done at the agent level. Because there are computational costs (transition functions) and communication overheads (message access) tied to each agent, determining an optimum partitioning strategy is not straight forward.

For example, optimising for a balanced memory utilisation and computational load by equally distributing agents across nodes can lead to excessive communication overheads due to all-to-all synchronisation of all message boards; grouping agents by type to reduce communication load can affect scalability due to uneven load and as well as limit the simulation sizes due to insufficient memory capacity in a heavily populated node.

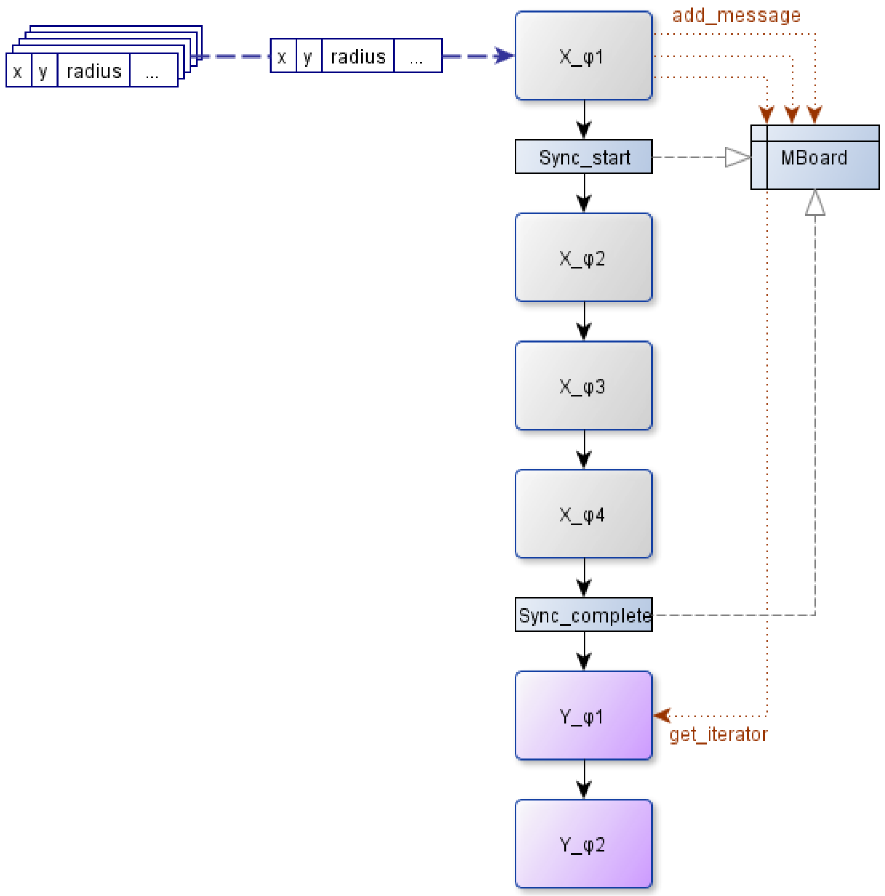


Figure 5: Independent tasks are scheduled during syncs process to maximise computation-communication overlap

### 3.2 Execution Path Bound by Model Definition

In the current framework, the memory access requirements of each transition function are not known to the framework. The parser therefore cannot make any assumptions about the actual dependencies between the functions and has to rely solely on the state diagram defined by the modellers.

More often than not, this leads to false dependencies between functions and an execution graph that is mostly sequential with very few concurrent paths. Such a graph would be tall and narrow, and expresses very little parallelism.

### 3.3 Thread Safety

A huge limitation of the current FLAME implementation is the lack of thread safety. Due to this short-coming, the framework is unable to safely execute multiple transition functions concurrently and is therefore less able to efficiently utilise multi-core systems. Users have to resort to launching multiple MPI task on a node to utilise all cores.

## 4 The Proposed Design

The following sections discuss some of the approaches we intend to explore in order to maximise the parallelism potential within the framework.

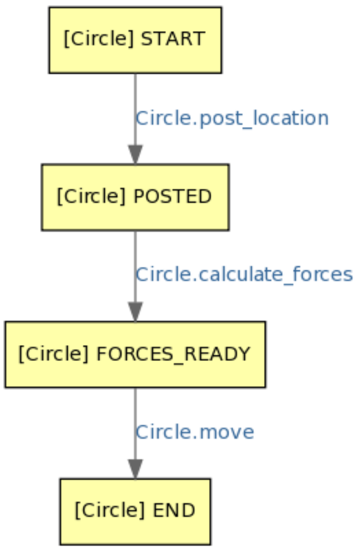


Figure 6: State diagram for model as defined by a modeller

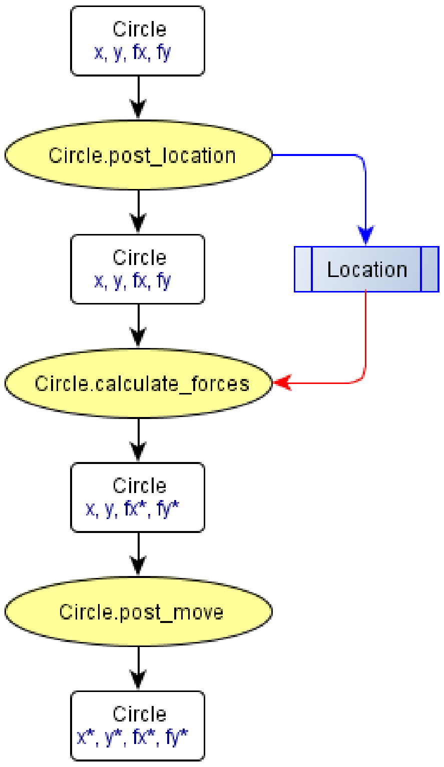


Figure 7: Function dependency graph based on states transition graph

## 4.1 Discovering More Parallelism Through Data Dependency Analysis

To improve the parallel performance of the framework, we need to extract as much concurrency as possible from a simulation. This involves breaking the simulation down into more parallelisable units then scheduling their execution in a manner which fully utilises all resources available to the execution environment.

We believe that the secret ingredient to achieving this is to have the memory access requirement of each agent transition function explicitly defined by modellers. With this additional information, along with the state transition graph of the agent, we can build a more accurate view of the dependencies between the different transition functions.

Every agent memory variable can then be treated as an independent entity and each write to the variable is seen as a transformation of the variable to a new version. Keeping track of memory reads and writes allows us to determine which functions can be run concurrently, and which ones need to be run in sequence to ensure that the correct versions of memory variables are accessed.

If a sequence of transition functions all read the same memory variables and never update the values, they are all reading the same version of data and therefore have no dependencies on each other. These functions can be executed concurrently (assuming there is no conflicting access to message boards). The same goes for functions which access different subsets of agent memory.

However, if a transition function require write access to a memory variable then subsequent transition functions are considered to depend on the new version of the memory variable and therefore have to wait till that information is available.

Once a data dependency graph is generated, we remove all data verticies while maintaining the implied dependencies between their parents and children by adding the children of each data vertex as a dependent of the parents. This reduces the graph to a directed acyclic graph (DAG)

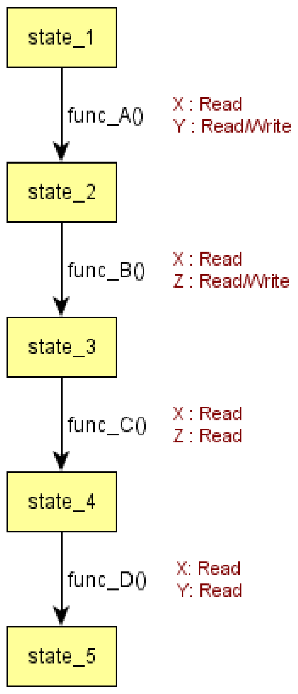


Figure 8: State diagram with memory access of transition functions explicitly declared

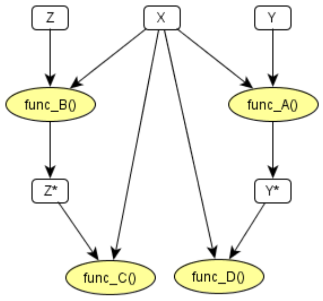


Figure 9: Data dependency graph of transition functions

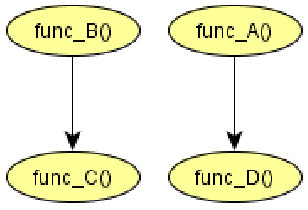


Figure 10: Function dependency graph derived from data dependency graph can express more parallelism

which represents the actual dependencies between functions; this graph is no longer bound by the state diagram produced by the modellers and can potentially express more parallelism.

Naturally, the amount of parallelism depends on the model. Take for example the Circles test model where each function consumes information produced by the previous function the resulting function graph is sequential as show in Figure 12 (the folder-shaped nodes in the figure – labeled `[Circle.x]` and `[Circle.y]` – represent potential I/O operations which store data vectors once they are no longer updated).

In contrast, a more complicated agent model with a richer set of actions and behaviour can better benefit from this approach. Take for instance the infection model depicted in Figure 13.

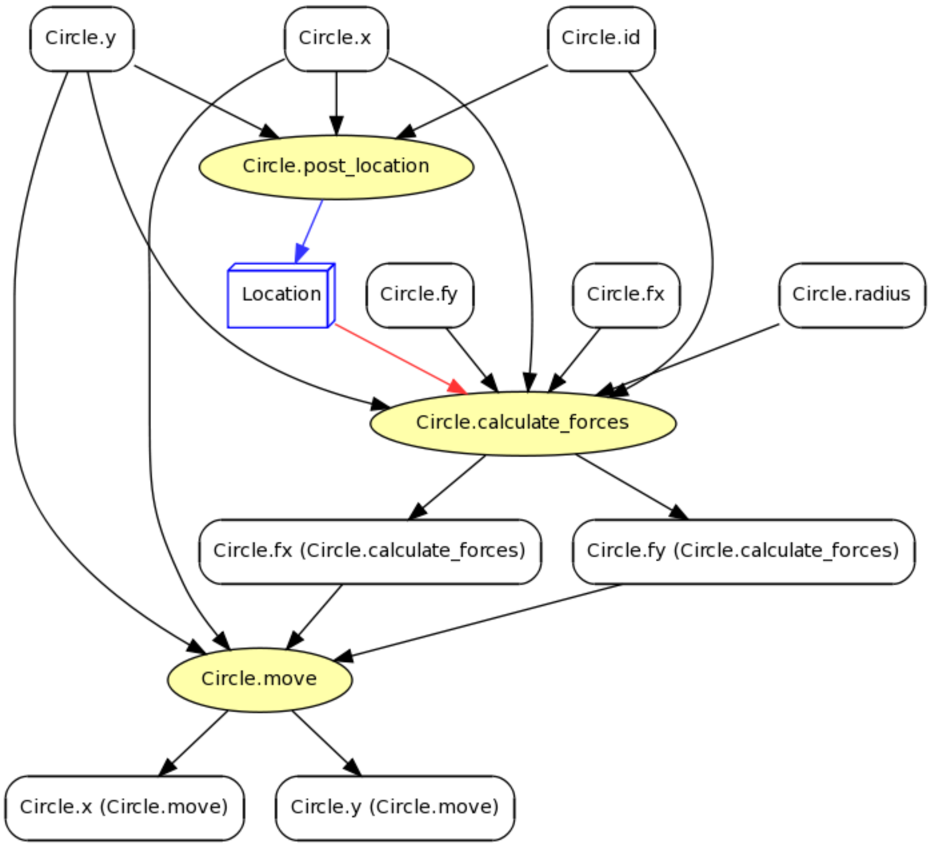


Figure 11: Data dependency analysis of Circles model

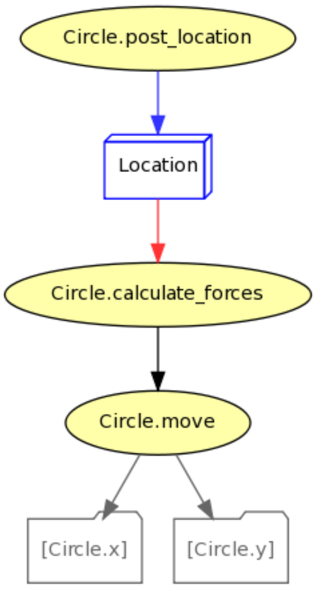


Figure 12: Function dependency graph of Circles model derived from data dependency graph

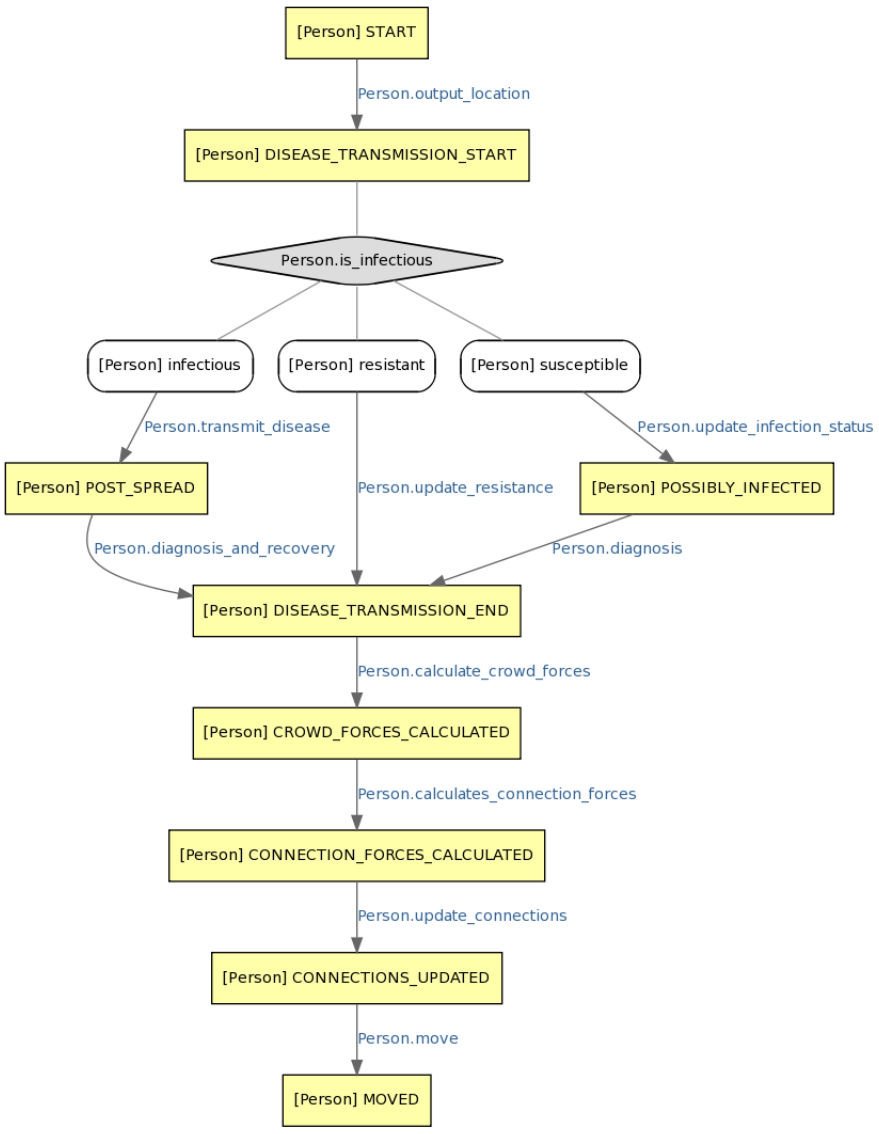

[Person] START
Person.output_location
[Person] DISEASE_TRANSMISSION_START
Person.is_infectious
[Person] infectious
[Person] resistant
[Person] susceptible
Person.transmit_disease
Person.update_infection_status
[Person] POST_SPREAD
Person.update_resistance
[Person] POSSIBLY_INFECTED
Person.diagnosis_and_recovery
Person.diagnosis
[Person] DISEASE_TRANSMISSION_END
Person.calculate_crowd_forces
[Person] CROWD_FORCES_CALCULATED
Person.calculates_connection_forces
[Person] CONNECTION_FORCES_CALCULATED
Person.update_connections
[Person] CONNECTIONS_UPDATED
Person.move
[Person] MOVED

Figure 13: State diagram of Infection model shows little parallelism

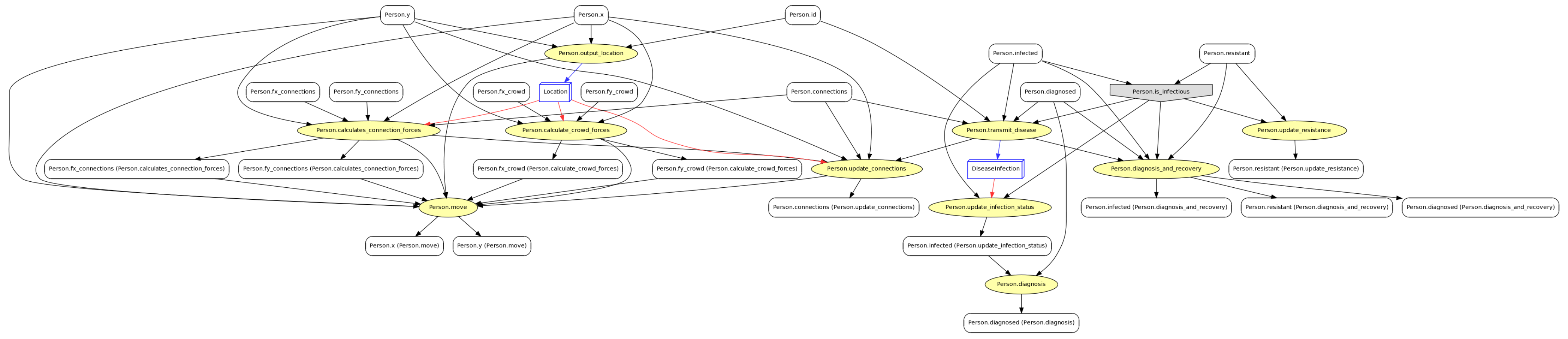

Person.y
Person.x
Person.id
Person.infected
Person.resistant
Person.output_location
Person.fx_connections
Person.fy_connections
Person.fx_crowd
Location
Person.fy_crowd
Person.connections
Person.diagnosed
Person.is_infectious
Person.calculates_connection_forces
Person.calculate_crowd_forces
Person.transmit_disease
Person.update_resistance
Person.fx_connections (Person.calculates_connection_forces)
Person.fy_connections (Person.calculates_connection_forces)
Person.fx_crowd (Person.calculate_crowd_forces)
Person.fy_crowd (Person.calculate_crowd_forces)
Person.update_connections
DiseaseInfection
Person.diagnosis_and_recovery
Person.resistant (Person.update_resistance)
Person.move
Person.connections (Person.update_connections)
Person.update_infection_status
Person.infected (Person.diagnosis_and_recovery)
Person.resistant (Person.diagnosis_and_recovery)
Person.diagnosed (Person.diagnosis_and_recovery)
Person.x (Person.move)
Person.y (Person.move)
Person.infected (Person.update_infection_status)
Person.diagnosis
Person.diagnosed (Person.diagnosis)

Figure 14: Data dependency analysis of Infection model

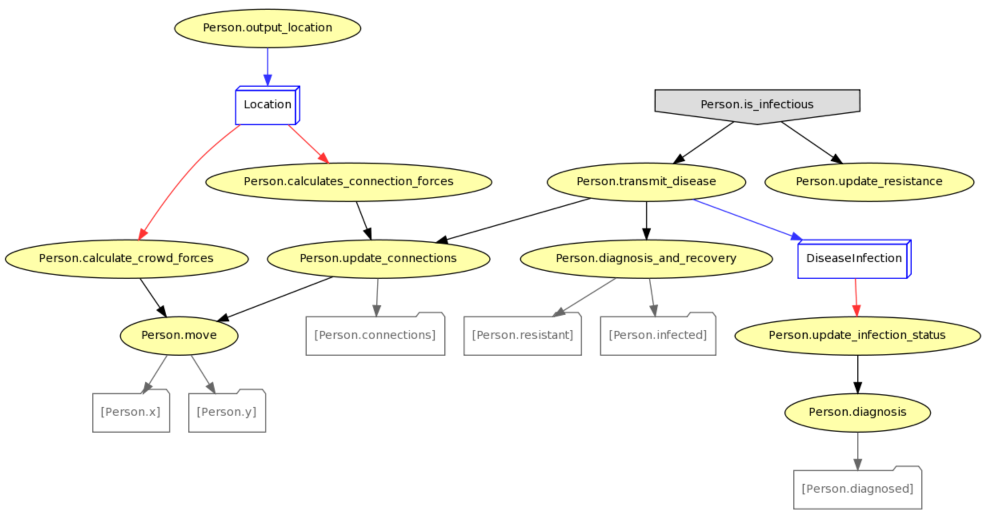


Figure 15: Function dependency graph derived from data dependency graph expresses more parallelism

We can see that the function dependency graph (Figure 15) derived from the data dependency analysis (Figure 14) is much broader than the state transition diagram. This graph expresses much more parallelism than one derived from the state diagram alone and will still produce the same results when used to drive the simulation scheduler.

## 4.2 Decomposing Agents into Independent Vector Operations

With the changes introduced in the previous section, transition functions can be treated as operations on a predefined set of independent variables. Since all the agents of the same type have the same set of transition functions and memory structure, we can effectively treat the transition function as an operation on long vectors where each vector element corresponds to an agent instance.

For example, the agent in the circles model which contain four memory variables can be represented as the following set of vector operations.

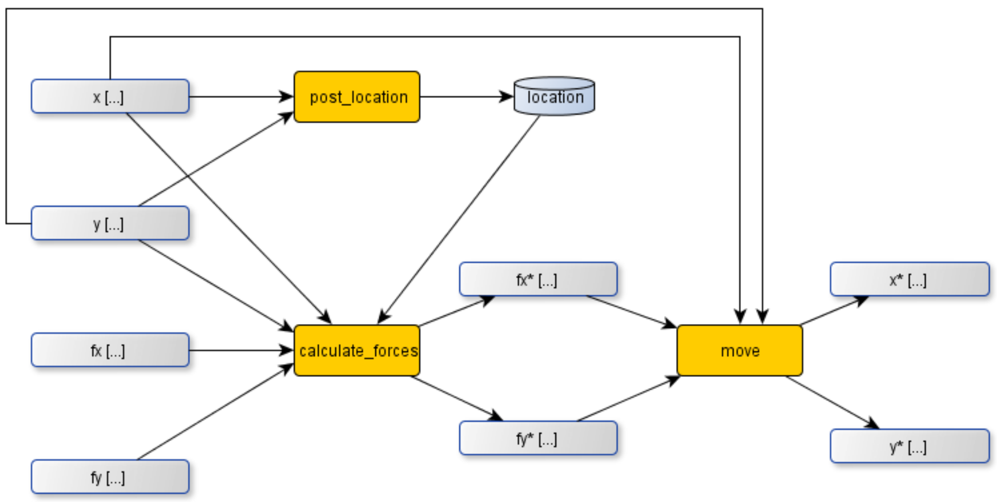


Figure 16: The Circles model represented as a set of vector operations

This shift in paradigm brings about many desirable features:

- The granularity of data has been reduced from an agent instance to a single memory variable. This allows us more flexibility in the storage structure and a more fine-grained approach to data and task decomposition.

- Operations on long vectors are potentially more efficient and can better utilise memory and caches. They are also more amenable to different parallel programming paradigms, e.g. SIMD, task farming, stream processing, etc.
- Check-pointing and migration of data can be done more efficiently – the elements in the long vectors are of equal size and contiguous in memory so data packing and buffering is no longer required. Furthermore, using information from the data dependency graph, data can be written to disk in stages as the final version becomes available.
- Transition functions can be treated as independent tasks that can be executed in any order as long as its inputs are available. This opens up many opportunities for optimisation including dynamic scheduling of tasks, multiple levels of parallelism, etc. (discussed in the following sections).

## 4.3 Dynamic Task Scheduling

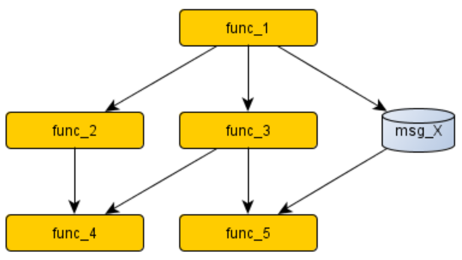
The function dependency DAG generated based on the analysis of memory reads/writes would implicitly encode the data dependencies. Therefore, as long as the function dependencies are met each function is guaranteed to be accessing the correct versions of memory and messages. This greatly simplifies the job of managing dependencies and ensuring the correctness of the simulation.

Instead of converting the DAG into a static sequence of function calls as done in the current FLAME framework, the DAG can be represented as a list of tasks to be consumed by a dynamic scheduler at runtime.

The use of a dynamic scheduler will allow the simulation to adapt to different runtime conditions and the variations in computational and communication loads that can occur in agent-based simulations.

Furthermore, a common runtime code can be used for all models which leads to less code generation for each model thus improving the testability and maintainability of the framework.

During a simulation, tasks are added to a queue as they become available (dependencies met) and the scheduler selects tasks for execution based on the priority levels assigned to each task. Once all tasks have been executed the iteration is complete and the whole process is repeated for the next iteration.



- `func_1 {T:agent, D:[], P:10}`
- `func_2 {T:agent, D:[func_1], P:4}`
- `func_3 {T:agent, D:[func_1], P:2}`
- `sync(msg_X) {T:msg, D:[func_1], P:50}`
- `func_4 {T:agent, D:[func_2, func_3], P:1}`
- `func_5 {T:agent, D:[func_3, sync(msg_X)], P:1}`

Figure 17: Task graph represented as a list of tasks

Each entry in the task list contains the following information:

- **Task identifier**: a unique handle for each task
- **Task type**: a label to determine which queue the task belongs to (more details later)

- **Dependency list**: list of tasks that must be completed before this task can be executed
- **Priority level**: the priority of this tasks

### 4.3.1 Using Task Priority to Optimise Resource Utilisation

The *priority level* indicates the urgency of each task. It assists the scheduler in determining which task from the queue should be executed first.

The priority level can be assigned based on many criteria, for example:

- **Sub-tree weight** – a task that has many dependents should be scheduled as early as possible.
- **Task type** – if there is only one queue, then the priority mechanism can be used to ensure urgent tasks such as message syncs are launched first and non-urgent tasks (such as data check-pointing) are only slotted in to fill the gaps.
- **Estimated run-time** – if profiling information is available from previous iterations, we can predict a task's runtime and weight it accordingly.
- **Vector length** – if profiling information is not available, we can use the size of the input vector length as an initial estimate for weighing the task. The priority levels should be recalculated periodically based on the runtime statistics collected during the previous iterations.

The job of recalculating the priority level can itself be wrapped up as a task that is managed by the scheduler. The same goes for other framework level operations that require significant use of available resources. This ensures that none of the resources are oversubscribed.

### 4.3.2 Using Multiple Queues to Manage Different Resources

Assigning a *task type* allows us the opportunity to support multiple task queues. Each queue can be assigned to different resources that can be managed independently.

For example, we may choose to have separate queues for disk I/O heavy tasks (for data check-pointing), communication tasks (message syncs), and computation tasks (execution of agent functions). In addition, different computation resources that can operate independently (CPU, GPUs, and other accelerators) can each have their own individual queue and be managed separately.

### 4.3.3 Using Slots to Control the Number of Concurrent Tasks Running on Each Resource

The scheduler queue is designed to ration the use of a particular resource type; there may be more than one instance of each resource (multiple CPU cores), or the resource may be able to handle several tasks simultaneously. To take this into account, each queue is assigned one or more execution slots which it can fulfil. The number of slots assigned to each queue will determine the number of concurrent task that uses a specific resource.

At runtime, the scheduler will attempt to maximise the use of resources by keeping every slot filled with running tasks, replacing each completed task with a new task from the associated queue. The total number of execution threads during a simulation would therefore be the number of slots plus the execution threads of the framework runtime (may be one or more).

Take for example a simulation running on 8-core nodes with a modest interconnect. A possible configuration would be to launch the framework with the following queues:

- *CPU* queue (7 slots): Up to seven CPU-bound tasks can be run concurrently at any time. This can be tasks associated to agent transition functions or framework related tasks such as the recalculation of task priorities or the indexing of message boards for faster lookups. One core is set aside (hence 7 slots and not 8) to handle the framework runtime as well as the processing requirements of other queues.
- *SYNC* queue (3 slots): Limit the number of concurrent message board syncs to avoid over subscribing the communication layer. Tasks from this queue perform mainly communication and require minimal computational resources.
- *I/O* queue (1 slot): Disk I/O to be done sequentially.

The types of queues and the number of slots for each queue should be user-configurable. In the absence of user preferences the framework should set sensible defaults based on the capabilities of the host machine. Tools such as *hwloc* [8] can be used to probe the capabilities of the host machine.

### 4.4 Adaptive Parallelism

When multiple slots are available, the scheduler has several options when mapping tasks to slots. The most straight forward choice would be to issue one task per slot; this approach will work reasonably well if there are many pending tasks in the queue. However, when the number of available slots exceed the number of pending tasks, some of the slots may be left idle while the remaining tasks get executed. This situation is particularly bad for models with bottlenecks in the function dependency graph where a huge portion of the execution graph depends on small number of tasks.

Another approach is to split a task across several slots each executing the same operation on different sections of the data vector. This method is most effective when the number of elements in the vectors is sufficiently large and the operation on each vector element takes a consistent amount of time – this ensures that the workload is evenly distributed across the resource and all instances of the task complete at the same time. In practice however, concurrent tasks almost never complete at exactly the same time. Waiting for all instances of the task to complete before issuing the next tasks makes the scheduler extremely vulnerable to load imbalance issues.

To benefit from both approaches, the scheduler should mix and match different strategies at runtime to adapt to the workload and available resource. Tasks can be scheduled to a single slot or scheduled across several (or all) slots. The following are some of the factors than can be considered when making the choice:

- The total number of slots
- The number of free slots
- The estimated completion time of running tasks
- The number of pending tasks
- The priority of pending tasks
- The estimated runtime of each task

Devising an effective scheduling algorithm that takes into account these factors will not be trivial and would make an interesting topic for further research.

## 4.5 Multi-Level Parallelism

With the changes proposed in this report, parallelism can be achieved at multiple levels each working on a different abstraction layer and can therefore be used together. The ideal combination of these different paradigms would depend on the characteristics of the model and the computational resource available.

### 4.5.1 Intra-Node Parallelism

With the introduction of scheduler queues and slots, all cores within a computing node can be used simultaneously to process tasks. In addition, attached processors/accelerators such as GPGPUs and FPGAs can potentially be thrown into the mix.

At this level of parallelism, only one instance of the simulation is running – parallelism is achieved by spawning one worker thread (or process) per slot and assigning tasks to them.

Scheduling tasks onto attached accelerators is trickier and would depend on the device itself. One approach we may consider is the use of OpenCL and its scheduler for offloading tasks onto attached devices.

### 4.5.2 Inter-Node Parallelism

To scale the simulation across nodes with distributes memory we shall take the standard SIMD approach of running multiple instances of the simulation on every node. Simulation data is decomposed and distributed across all instances and simulation is executed as usual. The model is coupled together through the synchronisation of the message boards – as long as agents have a consistent access to message boards, it shouldn't make a difference if the simulation is running as once instance or distributed across the cluster.

Important factors that affect the scalability of this approach are the synchronisation of the message boards and the decomposition of agent data. An ideal synchronisation strategy for the message board will hide all communication overheads and replicate as little data as possible; an ideal decomposition would minimise the need for message synchronisation and maintain a balance across all instances. Naturally, an ideal solution would be difficult to achieve due to the complexity and dynamic nature of agent models. Determining a decent strategy to maximise performance is non-trivial and would be another big part of this research.

### 4.5.3 Stacked Simulations

In the preceding sections we described how a model can be decomposed into a list of vector operations that is scheduled for execution based on a task dependency graph. Each vector operation is fed into a scheduler queue once its dependencies are met. The scheduler can launch one or more concurrent tasks depending on the number of slots available. Once all tasks have been executed the iteration is complete and the process is repeated for the next iteration.

To achieve good performance, all slots must be kept busy which means the scheduler queue must always be populated. For some models where there is a logical bottleneck (multiple operations depending on the output of a single operation or message) there is a high probability of the queue being empty when all remaining tasks depend on the task currently being executed in one slot, leaving other slots idle.

While it is sometimes possible to change the behaviour of the agent to avoid bottlenecks in the model, a change in model behaviour for the sake of performance is often undesirable or unacceptable.

To alleviate the problem of logical bottlenecks, we propose the use of stacked simulations i.e. merging multiple simulations and running them at once while maintaining the independence and output of each simulation. The multiple simulations can be of the same model with different input data, or different models, or even a combination of both approaches.

**Stacking Multiple Instances of the Same Model**

It is quite common for modellers to run a simulation multiple times for each experiment. These ensemble runs are necessary to deal with uncertainties in the system and may be done using the same input data for all runs or slightly different data for each run.

Instead of running each ensemble member sequentially or as multiple independent simulations, we propose merging the ensemble into a single FLAME simulation. By concatenating the data vectors for all runs and isolating the message boards of each simulation, we can simultaneously process all the runs using the same task dependency graph while maintaining the independence of each simulation.

While this does not solve the problem of logical bottlenecks in the task dependency graph, it greatly reduces the effects of it. The concatenated data vectors mean that each operation can be broken down into many more tasks which will keep the scheduler well populated. Furthermore, the additional computational load will give FLAME ample opportunity to handle all message board synchronisations in the background and effectively hide communication latencies.

It should be noted that this is only effective for bottlenecks on computational tasks. The effects of bottlenecks on communication tasks (message synchronisation) cannot be solved and may even be made worse by stacking ensemble runs.

**Stacking Different Models**

To properly mask problems with computation and communication bottlenecks, it is theoretically possible to merge the execution of different models under the same FLAME simulation.

In contrast to stacking simulations of the same model which simply extends the data vector processed by each task, stacking simulations of different models provides the FLAME runtime with an additional list of tasks that can be used to keep the scheduler busy. The simulations are effectively independent simulations with their own message boards and data vectors; they simply share the computational and communication resources via a common scheduler.

This form of stacking is very similar to the merging of nested models as implemented in the current version of FLAME with the exception that models should not interact with each other, and the namespaces of memory variables and message boards are kept separated for each model.

Both forms of stacking, be it of the same or different models, will be most effective when everything fits within the memory of a single shared-memory node. Once the memory requirements exceed that of one node and inter-node parallelism is required, its effectiveness would be down to how much additional communication is required and how well it can be masked by computational overlaps. It is therefore down to the model itself, and the decomposition strategy employed.

### 4.5.4 Coupled Simulations

Another opportunity which we could consider is the coupling of completely independent but related simulations, e.g. a flood simulation which provides the input data for an evacuation model. Instead of having to complete one simulation and then feeding the results as input to the other model, both (or several) models can be run simultaneously as separate instances.

To couple the models together, a shared message board is used. This message board will behave as a standard message board but will use an AMQP [9] (Advanced Message Queuing Protocol) backend for inter-simulation communication.

The same mechanism can be used for multi-scale models – multiple micro-level models can be running independently all of which interact with a parent model that simulate the model at a macro level.

In contrast to the nesting of sub-models (available in the current FLAME framework) or the stacking of multiple models (discussed in the previous section), this form of model integration is very loosely coupled. Each model will have its own timescales and can progress asynchronously; blocking only happens when a required message from a remote simulation is unavailable and performance is only affected if the scheduler does not have other tasks to process in the meantime.

Using this scheme, the key to good performance would therefore be to calibrate the resources assigned to each model such that the production and consumption of inter-model messages occurs at roughly the same rate.

### Other Uses of AMQP-Based Message Boards

Using a messaging protocol such AMQP as a synchronisation back end for the shared message board, communication is no longer limited to a group of processes started in an SIMD fashion (as with MPI). Any client connected to the message broker can participate in the communication, therefore interaction via these boards is not limited to FLAME models.

- Existing applications can be modified to communicate with a FLAME model using a pre-defined API. This allows the coupling with non-FLAME simulations and external solvers.
- A mock application can be easily written to feed data to boards. This is useful for running the model in a standalone fashion and is especially useful during the development and testing of the models.
- Remote visualisation and computational steering can be achieved by simply reading and writing to the shared message boards from a desktop GUI.

## 5 Design Patterns for Parallel Agent-Based Models

When moving from an inherently serial agent model to a one designed for parallel execution such as FLAME, many interaction patterns that occur frequently in agent models take on a new level of complication. Access to globally shared information such as environment data and location of all agents can no longer be taken for granted as any update to global information has to be synchronised across distributed processes before they are accessible by other agents.

Take for instance a simple example where agents move around a grid by randomly selecting a new location that is not occupied by another agent. In a serial implementation, each agent is activated in a random sequence and it chooses a free location to move to; every move updates the global information such that the next agent that is activated is aware of that new location. To achieve the same behaviour in a parallel model would require lock-stepping the activation of agents across all processors and synchronise the global information after each move, which would be a prohibitively expensive approach and defeats the purpose of parallelism.

One possible solution to that problem is for all agents to propose a move (in the case of FLAME, by posting a message) based on a snapshot of the current global information and subsequently make the move if there are no conflicts with other proposals. Conflicting proposals can be dealt with in various ways – by having the agents stay where they are, or assigning a random number to each move proposal and using it choose who wins, or by going through another round of proposals, or anything else a modeller can come up with. Each choice of conflict resolution strategy will have an effect on the outcome of the simulation and possibly the performance of the model. Complications such as these arise frequently in the models weve come across and this greatly contributes to the steep learning curve of building an efficient parallel agent based model.

Quite often, similar solutions to these problems are transferrable to models from different domains. For instance, the same interaction pattern can be used to model pigeons feeding on fields with finite amount of food in a population dynamics model, as well as households purchasing goods from stores in a consumer goods market model. These problems have been solved by modellers using ingeniously crafted interaction patterns between agents; however the implementation is coded within their models and not readily searchable or transferrable to other models.

We believe an important step towards improving the usability for FLAME and helping modellers write efficient models is to identify, categorise, and document reusable solutions for parallel agent-based modelling (similar to the popular Design Patterns [10] for Object-oriented program design).

Once a set of reusable solutions is defined, we can proceed to implement framework-level solutions that modellers can quickly integrate into their models making it easy for new users to get their models up and running as well as to quickly evaluate different approaches to their problem.

## 6 Handling Agent Topology

FLAME currently has no notion of agent topology[1] or message semantics. All agent interactions rely on a generic message board that acts as a medium for all-to-all messaging. Filters were introduced to emulate one-to-many (broadcast) and one-to-one communication. This allowed for a reasonably efficient parallel implementation of a general purpose agent modelling framework.

Agent topology (e.g. Euclidean space, grids, agent hierarchies and networks) is currently implemented within the models themselves. For example, some notion of agents populating a 3D Euclidean space can be implemented by storing positional information in agent memory and conveying that information to all other agents using a `Location` message board.

The need for modellers to reinvent the wheel each time raises the barrier to adopting FLAME especially when a shift in mindset is required to design an inherently parallel model (no direct access to global space). Models often end up more complicated than is necessary making it harder to test and more prone to errors.

In this reengineering effort, we will take a serious look at integrating agent topology into the framework. As with the case of common agent interaction patterns, providing a framework level solution to agent topology will be much more efficient and will improve the usability of the framework.

As a starting point, we believe an agent topology manager should provide the following components:

- Mechanism to initialise a topology (in model definition) and register agent instances to inhabit that topology (in model definition or initial data).
- Topology-specific message boards which expose relevant filters and routines, e.g. return only messages from neighbours. Iterators for these boards will be optimised for the relevant data types and query methods.
- Pre-build solutions for common topology-specific agent interactions, such as establishing a new link in a graph or avoiding conflicts when moving to a free cell in a grid.

In should be noted that an efficient parallel implementation of an agent topology manager has to take into account the decomposition of agent data across distributed memory nodes. In turn, the agent topology can be used to influence the decomposition of agent data in order to reduce inter-node communication.

Other factors such as the use of multiple topologies in a model, and the memory model of the execution environment (share memory or distributed memory) also have a huge impact on the implementation. Providing a canned solution for all potential users is therefore impractical. A more scalable solution would be to provide a common frontend for the different topologies, and

[1]Not completely true. The spacial partitioning routine inherited from the previous incarnation of FLAME will look for specific agent memory variables (x, y, z) to determine its location in a 3D Euclidean space.

a modularised backend to allow advanced users to customise the inner workings to suit their model and execution environment.

Further investigation into different use cases and user requirements is necessary before we can finalise the design.

## 7 Backward Compatibility

We acknowledge that backward compatibility is an important issue especially for existing users who have invested significant effort in their models. However, considering the scope of the reengineering planned, it is unlikely that direct backward compatibility will be possible. It is therefore important that the migration from the old to new framework be as easy and painless as possible for users.

To address this, a backward compatibility analysis is required; this will be an ongoing process that runs in parallel with the development of the new framework and will include the following activity:

- List existing features and syntax
- Evaluate each feature and identify those that will be deprecated or changed
- Determine the impact of each deprecation/change and identify a migration path
- Determine the impact of new features/syntax and identify migration path

The following are the possible migrations paths sorted by preference (starting with the preferred option):

1. **Automatic translation by the framework** – running the old model in the new framework without change. The intended behaviour of the model should not change and sensible defaults should be selected for information that is not expressed in the old model. A warning message should be displayed (along with porting information) if a deprecated functionality is ignored or if a change in the model is needed to improve performance.
2. **Tool-driven transformation of model** – transformation of an old model to a newer format using a transformation tool provided by the framework. This should be automated where possible and require minimal user input. As with the previous approach, users should be warned of deprecated features or if changes are required to improve performance.
3. **Guided migration** – fully documented steps and guidelines for users to replace deprecated functionality. This document should also include recommendations for users to optimise their model and better utilise the new framework.

## 8 Conclusion

In this report, we have described the current design of FLAME and its limitations. We then proposed an architecture that decomposes the simulation into a list of vector operations that can be scheduled based on a dependency graph. The dependency graph can be generated by analysing the memory accesses of each agent function.

The new approach will enable various optimisation opportunities including more efficient data structures, better resource utilisation using dynamical task scheduling, and multiple levels of parallelism. Each of these was briefly discussed with varying levels of detail.

Other important considerations such as efficient handling of agent topologies and complex interactions as well as managing backward compatibility were also discussed.

## 9 Acknowledgement

This work has been funded by EPSRC Grants EP/I030654/1 and EP/I030301/1.

## References

[1] FLAME Website - `http://www.flame.ac.uk`

[2] S. Adra, T. Sun, S. MacNeil, M. Holcombe and R. Smallwood, "Development of a three dimensional multiscale computational model of the human epidermis", PLoS ONE 5(1): e8511. doi:10.1371/journal.pone.0008511

[3] T. Sun, P. McMinn, S. Coakley, M. Holcombe, R. Smallwood and S. MacNeil, "An integrated systems biology approach to understanding the rules of keratinocyte colony formation", J. R. Soc. Interface 22 December 2007 vol. 4 no. 17 1077-1092

[4] Holcombe et al., "Modelling complex biological systems using an agent-based approach", Integr. Biol., 2012,4, 53-64

[5] C. Deissenberg, S. van der Hoog, and H. Dawid, "EURACE: a massively parallel agent-based model of the European economy," Applied Mathematics and Computation, vol. 204, no. 2, pp. 541552, October 2008.

[6] T. Balanescu, A.J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan (1999), "Communicating stream X-machines systems are no more than X-machines", Journal of Universal Computer Science, vol. 5, pp. 494507, 1999.

[7] L.S. Chin (2009) "libmboard Reference Manual (Version 0.2.1)", October 2009.

[8] Portable Hardware Locality (hwloc) - `http://www.open-mpi.org/projects/hwloc/`

[9] Advanced Message Queueing Protocol (AMQP) - `http://www.amqp.org`

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides (1995) "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley