Manchester
Metropolitan
University

Combining Constraint-Based and Imperative Programming in MABS for More Reliable Modelling

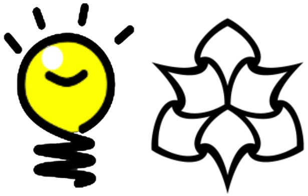
Bruce Edmonds,
Centre for Policy Modelling
(long-term work in parallel to Gary Polhill,
James Hutton Institute)

Motivation

Did you ever do any of the following?

- Write some fairly arbitrary code because, although some characteristics of the target process are known, many details are not
- Use a random generator to add noise into a simulation and then average the results, hoping this gives you a good idea of outcomes
- Fail to add in sufficient internal checking and tests because it messes up your code
- Reimplement standard entities, not knowing if you did this in exactly the same way as others

Did you ever do any of the following?



- Implement different aspects of a simulation based on accounts by domain experts, but with no formal check on their consistency
- Import a sub-model that someone else developed and tested, but had no check it was working correctly
- Look at someone else's simulation and be unclear which parts are essential core assumptions and which are more contingent implementation choices

Some Relevant Existing Work to Generalise From

Strong Type Checking

- Constrains variables to only be able to contain the right kind of data
  - e.g. a list of strings, a positive integer
- Checks both syntactically in the code as well as during run time
- Prevents some subtle bugs
- The information is often embedded within the code and hard to separate out
- Limited in what checking it can do

Stubs

- Temporary code one puts into a model so you can program and test other parts
- Often specifies some null or random behaviour
- Needed with complex models with different interrelated parts
- Later revisited and fleshed out
- This is a matter of degree, often MABS have stub-like elements for some aspects
- Some are just left in the code as they are!

Random Number/Choice Generators

- Often used when:
  - We do not understand how to code the particular behaviour
  - In stubs, as a temporary measure
  - As “noise” to explore robustness etc.
  - We think the process does not matter to results
- But these only sample possibilities
- Different kinds of generator have their own properties which may drive results

Formal Ontologies

- Defines the agreed entities (both abstract and observed) with a set of labels
  - e.g. money, person, household, field, norm…
- Defines the important, definitional relationships between these
  - e.g. a household is a set of people, a farm is a farmer plus a set of fields
- There are tools to formalise these logically
- Difficult to get agreement upon these!
- It was hoped useful things could be inferred from these, but such inference was ‘thin’

APIs

- A precise description of how a unit of code (agent, module, procedure) should interact with the rest of the code
- Agreed between people/tasks so these can be separated out
- If something abides to its API one does not have to understand how it works inside
- Checking some code does abide to its API is difficult and time consuming

Internal and Unit Tests

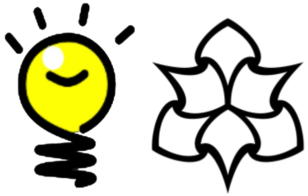
- In more complex simulations there is always a danger of hidden bugs/mistakes
- Thus, it is good practice to program in lots of internal checks to catch some of these
  - e.g. that there are never any negative prices
- Unit tests are a suite of tests that checks the standard behaviour of some code, which can be run when the code is changed
- Once assured they can be switched off
- Can be embedded in the code, so hard to find

Integrated but Seperable Declarative and Imperative Layers

Declarative vs Imperative Programs

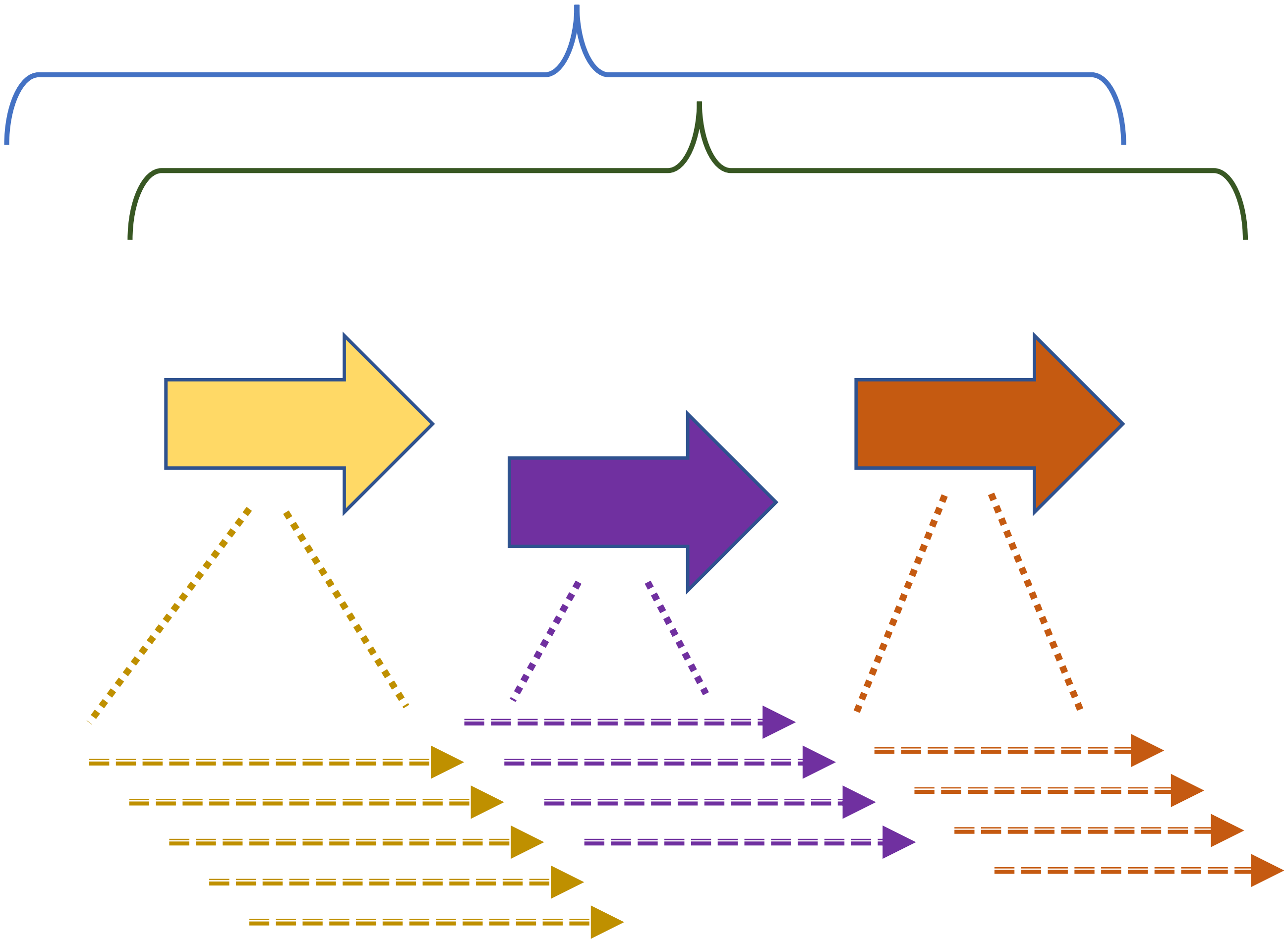
- In imperative programs (NetLogo, python, R…) one specifies (at the micro-level) what steps in a process should be done in turn – the computer then does these in that order
- In declarative programs (Prolog...) one writes a set of statements, specifying states and relationships, and the computer finds a computation that is consistent with these
- In constraint programming this is optimised for combinotronic problems

The Main Idea



- Separable declarative and imperative “layers” describing the *same* simulation
- Declarative layers may include standard specifications decided by community
- Early execution using declarative layer with minimum of generators added
- Gradually more imperative details added in (where these are known)
- When all is checked, imperative layer only can be executed for speed/sensitivity etc.

Different “Layers” of a Simulation



Declarative Statements

Alternative Imperative Implementations

Simulation Results

Possible Programming Sequence

1. Ontology
2. Key assumptions
3. Extended checking
4. Generators and fillers
5. Add in imperative details gradually
6. Turn off declarative layer for speed, sensitivity analysis

Advantages of the approach

- Joining models
- Comparing models
- Enhanced error checking
- Reusable modules/specifications
- Rapid prototyping
- Support more automation of modelling
- Separation of specification and implementation
- Possibility of exploring all behaviours that satisfy the constraints (albeit inefficiently)
- Declarative layer may include information to inform efficient meta-computation (parallelism/surrogates)

Disadvantages of the approach

- We do not yet have such a system
- More technical machinery to get one's head around
- Purely declarative execution will be slow
- Spoiling modellers' fun by making modelling more of a formal process
- To leverage the most advantage from this architecture the community needs to agree on some declarative specifications

Some Previous Work

- A strictly declarative modelling language, SDML (Moss & al. 1998, Edmonds & al. 2002)
- …and its use for constraint exploration (Terán et al. 2001, 2002)
- Importance of ontologies for socio-ecological system modelling (Polhill & Gotts 2009, Gotts & al. 2019)
- A NetLogo extension for extracting an ontology from a model (Polhill 2015)
- A declarative simulation architecture – “OBIAMA” – based on ontologies (Polhill 2016)

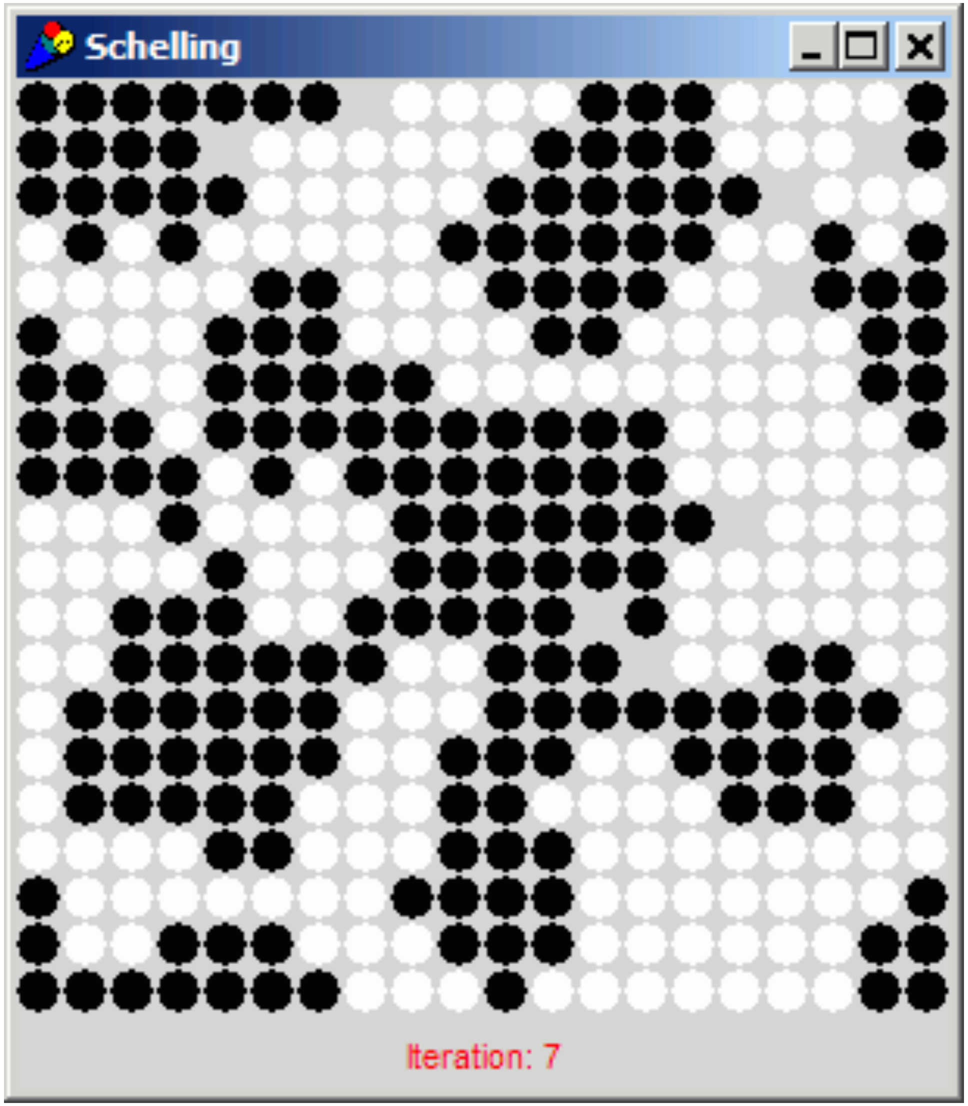
An Example

The Sakoda/Schelling Model (again ☺)

Setup: *agents*, which are in *neighbourhoods* and have a *property*, some spaces left empty.

Dynamic Rule: repeatedly: each agent looks at its *neighbourhood* and if *not sufficiently similar to self*, it *moves to an empty square*.

Key Conclusion:
Segregation can result from wanting only a few similar neighbours



The *specific* assumptions in the standard 2D Schelling model…

- Agents have one of two colours
- Fixed equal numbers of agents with each colour
- Leaving some spaces
- Randomly placed at set-up on a 2D grid
- Neighbourhoods = Moore Neighbours
- Sufficiently similar = proportion of other agents in neighbourhood with same colour as self > a parameter
- If moves, destination = random empty square
- Synchronous update (all agents check each time click)
- Segregation = average proportion of neighbours are same colour as self over all agents

Declarative Entity Specification

- Entities: “locations” with property and agent-occupier/empty
- Entities: Agents with a property and location
- Constraint: each location has at most one agent with its location = self
- Constraint: where agents think they are matches what occupiers locations think they have
- Constraint: there are always some empty? locations

Dynamics

- Dynamic function: Satisfaction(agent) → Boolean = predicate(properties of (agents@neighbourhood(location(agent))))
- Constraint: predicate(all properties same as self) = true
- Constraint: predictate(no properties same as self) = false
- Update process for an agent:
  - if not satisfaction(agent) then
    - next time: location(agent) ← one of empty locations
    - Next time: that location set to not empty?

Space is Complex!

There are different ways of defining neighbourhoods/space, each with subtle consequences! For example define…

1. given a network of links between locations, neighbourhoods are all locations N links, or less, away
2. given a distance metric between locations, neighbourhoods are all locations < set-distance away
3. given a distance metric between locations, neighbourhoods are N locations closest in distance
4. any of the above then randomly add 10% of agents into an otherwise distant neighbourhood

Defining Neighbourhoods

- Locality seems important in this model, but intuitions about space include many aspects
- Fixed neighbourhood function: location → subset of locations
- But which properties of neighbourhoods are essential for the stated results?
  - Constraint: dist(A→B) + dist(B→C) >= dist(A→C) (triangle inequality) ??
  - Constraint: many of your neighbours’ neighbours are your neighbours (clustering) ??

Measuring Segregation of Model State

- Function: seg(model) → [0, 1]
- Constraint: If all agents only have own property in neighbourhoods, seg(model)=1
- Constraint: seg(model) >= seg(model’), where model’ has two agents swapped so that the number of links between similar agents increase
- But lots of specific ways of measuring this!

Possible Generators for Exploration

Different (combinations of)…

- neighbourhood functions
- kinds&levels for the satisfaction predictate
- distinct kinds of property
- numbers of each kind of agent
- movement algorithms
- number of empty spaces
- initial locations

Specific Implementation

- If outcomes using a particular generator rarely make much difference then, replace it with a specific, fast algorithm
  - e.g. initial locations decided randomly
- If any are known or determined by target problem, replace generator with appropriate algorithm
  - e.g. if social structure of links is known, use that

What have we achieved?

- Separated out the declarative specification and the imperative code…
- …which can be the more essential assumptions and the more specific ones
- It allows a more incremental implementation approach with more checking included
- The declarative specification can be published separately so others can use it to:
  - explore the space of possibilities, knowing it is the same space
  - try their own imperative variants (and if they need to change the fundamentals, this triggers a different kind of debate)

Rather than 1001 varieties…

We might be able to…

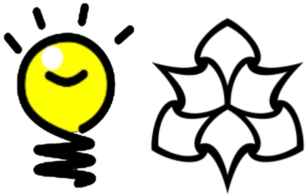
- find what general properties are essential to get target results and thus know when it might be applicable to observed cases
- avoid specific implementations that have bugs or are very brittle (require fine tuning for results)
- know how to standardise either declarative properties or implementations of parts (like movement or neighbourhood)

Conclusions

Hope of Better Reliability

- Better, and more flexible, internal checking
- Separable declarative descriptive & checking layer that can be verified by others
- Earlier proof-of-concept enables to check the direction with others (e.g. stakeholders)
- Separation of code representing core assumptions with implementation specifics
- Clearer theory development with necessary conditions for results established

Facilitating working as a Community



- Standard definitions of entities and relationships can be developed
- Declarative specifications may help check compatibility of models when joining them (and so help avoid “integronsters”)
- It may allow libraries of model components to be more reliable and easier to find (by some automation of which will work) and thus help with modelling for crises
- Interaction about abstractions and specifications can happen separately to those of implementations – e.g. aiding tool add-ons

Some Relevant References

Edmonds, B. and Wallis, S.: Towards an Ideal Social Simulation Language. *3rd International Workshop on Multi-Agent Based Simulation (MABS'02)*, Bologna, July 2002. LNAI, vol. 2581, pp. 104-124. Springer. (2002).

Edmonds, B.: How Formal Logic Can Fail to be Useful for Modelling or Designing MAS, In *Regulated Agent-Based Social Systems*. LNAI, vol. 2934, pp. 1-15. Springer. (2004).

Galán, J.M., Izquierdo, L. R., Izquierdo, et al.: 'Errors and Artefacts in Agent-Based Modelling'. *Journal of Artificial Societies and Social Simulation,* 12(1),1. https://www.jasss.org/12/1/1.html. (2009).

Giannesini, F., Kanoui, H., Pasero, R., and Van Caneghem, M.: *Prolog*. Addison-Wesley. (1986)

Jaffar, J. and Lassez J.L.: Constraint Logic Programming, in Proc. *The ACM Symposium on Principles of Programming Languages*, ACM, (1987).

Moss, S., Gaylard, H., Wallis, S. and Edmonds, B. (1998). SDML: A Multi-Agent Language for Organizational Modelling. *Computational and Mathematical Organization Theory*, 4, 43-69.

Parker D.C., Polhill J.G. and Mussavi Rizi S.M.: An OWL (Web Ontology Language) representation of the MR POTATOHEAD agent-based land-use change meta- model. Presentation to the American Association of Geographers Annual Meeting, Boston, April 15-19, (2008).

Polhill, J. G., & Gotts, N. M. (2009) Ontologies for transparent integrated human-natural system modelling. *Landscape ecolog*y, 24, 1255-1267.

Polhill, G., & Gotts, N. (2010). Semantic model integration: an application for OWL. European Social Simulation Association conference, 19-23.

Polhill, G., & al. (2012). An ontology-based design for modelling case studies of everyday proenvironmental behaviour in the workplace. 2012 International Congress on Environmental Modelling and Software, Leipzip, Germany. https://scholarsarchive.byu.edu/iemssconference/2012/

Polhill, J. G. (2015) Extracting OWL Ontologies from Agent-Based Models: A Netlogo Extension. *Journal of Artificial Societies and Social Simulation*, 18(2), 15. <http://jasss.soc.surrey.ac.uk/18/2/15.html>

Squazzoni F., Polhill J.G., Edmonds B., et al.: Computational models that matter during a global pandemic outbreak: A call to action. *Journal of Artificial Societies and Social Simulation*, 23(2), 10. DOI: 10.18564/jasss.4298. (2020).

Stepney, S., and Polack, F.A.C.: *Engineering Simulations as Scientific Instruments: A Pattern Language*. Springer, (2018).

Terán, O. and Edmonds, B.: Constraint Model-based Exploration of Simulation Trajectories in a MABS Model. CPM Report 06-161, MMU. (http://cfpm.org/cpmrep161.html). (2004).

Terán, O., Edmonds, B. and Wallis, S.: Determining the Envelope of Emergent Agent Behaviour via Architectural Transformation. *7th International Workshop On Agent Theories, Architectures And Languages* (ATAL2000), Boston, MA, 8th-9th July, 2000. LNAI, vol/ 1986, pp. 122-135. Springer. (2001).

Voinov, A., & Shugart, H. H.: ‘Integronsters’, integral and integrated modeling. *Environmental Modelling & Software*, 39, 149-158. (2013).

Moss, S., Gaylard, H., Wallis, S. and Edmonds, B. (1998). SDML: A Multi-Agent Language for Organizational Modelling. Computational and Mathematical Organization Theory, 4, 43-69.

Schelling, Thomas C. 1971. Dynamic Models of Segregation. Journal of Mathematical Sociology, 1:143-186.

Sakoda, Jaames M.(1971) The checker board model of social interaction. Journal of Mathematical Sociolog, 1:119–132(1971).

Thanks!

Manchester
Metropolitan
University

These slides are at: http://**cfpm.org/**slides
as BE-MABS-2023.pdf *(or use this QR code)*