A Computational Platform for DT Models

Carlo Jaeger, Steffen Fürst, Manfred Laubichler, Sarah Wolf*

June 28, 2020

In the coming years, the models used in DT (Decision Theater) events shall allow for and indeed foster cumulative developments of knowledge. In the context of complex adaptive systems, sustainability, innovation, and global futures, the models should represent a large variety of agents and the material as well as ideational environments they interact in. Participants in a DT event shall interact with models where they can assume the roles of decision makers, shaping the behaviour of "focal agents" in the models and viewing effects of the decisions that these focal agents can observe.

Cumulative development of knowledge in DT work calls for a computational platform that allows to represent in a standardized way the different agents and entities involved in very different models. A key requirement for such a platform is that it should allow simulations to be run on machines with vastly different computing power. When used with high performance computers, the platform must allocate the available computational resources to different model components depending on the computational cost each element generates – a cost that typically varies in the course of a simulation run.

These requirements are fulfilled by the GCF computational platform for DT models.[1] This text introduces version 0 of the platform. A separate tutorial displays and explains the code relevant for professional users and shows how to use it.

1 DT model features

Models of complex human-environment systems – including social, economic, ecological, technological components and interactions – must be able to include combinations of the following features:

- agents can be of different types, e.g. governments, citizens, households, firms, animals, microorganisms, etc., and the model representation of agents may draw on existing theories and modelling conventions. Economic agents, e.g., may or may not be expected utility maximizers, or an agent may be a network of agents at another scale, as considered in extended evolution. Agents may disappear and new agents may emerge in the course of a simulation
- for use in DT events, some agents must offer the possibility to be taken over by the participants of such events
- agents can exchange information, goods, etc. with each other in complex network structures that may co-evolve with the agents
- agents are embedded in a common environment that can be spatially differentiated and may co-evolve with the agents
- agents, but also networks and environments, may be extensively data-based, e.g. the set of agents can be a synthetic population that matches statistical distributions of the real-world population for relevant features

*Comments are welcome to sarah.wolf@globalclimateforum.org.

[1]The GCF computational platform for DT models, version 0, has been developed by Steffen Fürst with support from Sarah Wolf, Gesine Steudle, Manfred Laubichler and Carlo Jaeger. They all work with the Global Climate Forum, Sarah Wolf at the Biocomputing Group, Institute of Mathematics at Freie Universität Berlin, Manfred Laubichler and Carlo Jaeger at the ASU Global Futures Laboratory. Current work on the platform builds on research performed by GCF – with massive contribution by Andreas Geiges, now at Climate Analytics – in an EU center of excellence for harnessing high performance computing to study global challenges (see http://coegss.eu).

- input representing decisions of DT event participants can be incorporated in the model, and output can be visualised in interactive response time
- the dynamics of the system are generally not deterministic

Later on, these model elements and components will be more closely specified, constructing, as it were, a "DT modelling platform" on an abstract level. However, this is beyond the scope of the present text, that focuses on the computational platform instead. The relation between the two may be understood in analogy with the relation between a general purpose programming language and a domain specific language.

2 Computational platform features

To accommodate the mentioned features of DT models and allow for parallelization of simulations, the GCF computational platform implements an **extended and generalized parallel graph dynamical system**.[2] With such a structure, efficient load balancing in parallel runs can be achieved by graph partitioning. This puts a particular focus on the network structures between agents and other entities in the modelled system, which is not the case in well-established parallel ABM frameworks[3];

In a simple graph dynamical system, the underlying graph is fixed, vertices have states in a common state space, there is a transition function for each vertex that computes its next state from the current state of the vertex itself and its nearest neighbours in the graph, and an update scheme provides the mechanism by which these transitions are composed. This is generalised here as follows.

Nodes are called **computational agents**, they may be of **different types**, e.g. representing persons, households, firms, etc. and have a state that belongs to a type-specific state space. Computational agents may or may not represent agents in the modelled system; in particular, it may take more than one computational agent to represent a modelled agent. **Edges** between the agents are **directed**, if a transition of agent a depends on the state of agent b, an edge from b to a is needed. Edges may also have different types (sometimes one speaks of a coloured graph; here, different colours represent different kinds of interactions, e.g. a network for goods exchange may differ from a network for information exchange) and may have a state (e.g., when representing an exchange of goods, an edge's state may report the quantity).

The main graph for the interactions between agents is extended by two additional layers; one for spatial locations, and one for specific computational entities that need edges from and/or to (almost) every computational agent in the main interagent graph. The latter may be the case, e.g., for model parameters that all (or large sets of) computational agents get as input, for gathering information from them to generate model output for visualisation, and for aggregating information within the model that is passed back to computational agents. As for the spatial dimension of models, the spatial layer provides a discrete grid (in mathematical terms a matrix). Each element of the grid can have a state and transitions, too, and each computational agent can, but need not, be assigned to an element of the grid. In particular, elements of the grid may also play the role of aggregating information from all agents in the explicit graph "located" there.

[2] While many platforms for the development of ABMs exist – see `https://www.comses.net/resources/modeling-platforms/` for an overview – only a few of them allows to run a parallelized simulation on multiple nodes of a computer cluster. Moreover, when they support spatially explicit models out of the box, load balancing is usually based on a rectangular decomposition of the spatial domain, which is problematic e.g. for models with unequally distributed population density.

[3] `https://evoplex.org/` is the only exception known to the authors; unfortunately the documentation for the model development is not very extensive. `https://www.informs-sim.org/wsc11papers/025.pdf` presents InterSim, a general-purpose graph dynamical system modeling framework, but the code is not publicly available and it is not clear whether it is still used.

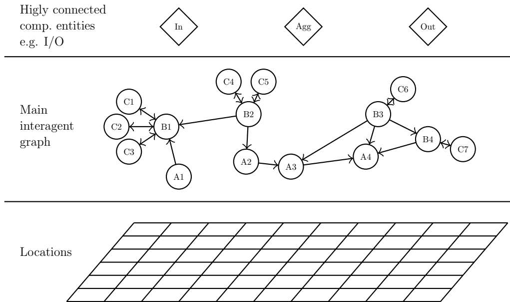

Fig. 1: Main interagent graph with additional layers

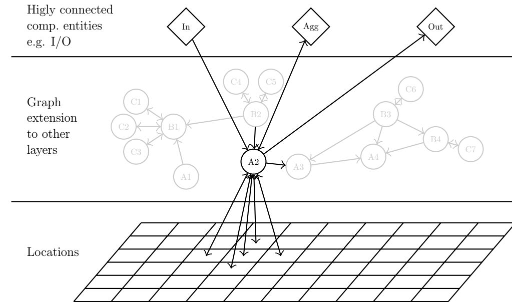

Fig. 2.: Explicit and implicit edges for an example computational agent

To model the system's dynamic evolution, we consider **discrete time steps**. At each step t, there is a set of computational agents; with each step, new ones may be added or some existing ones may be removed. Similarly, the set of edges may evolve. For all agents and possibly for the edges present at the beginning of a simulation, an initial state must be given.

In a transition, an agent's new state at time t is then computed based only on information from the previous step. Notice, however, that the state of an agent may include memory of previous steps. Information from step $t-1$ usually includes the previous state of the agent itself, and may include states of adjacent agents in the explicit graph, possibly also the states of the respective edges themselves, as well as adjacent entities implicitly linked to it by edges of the right direction. This can be interpreted to represent the convention that the state of an agent cannot be changed by another agent; in fact, for clarity, the state is recreated at each step, not merely modified. The underlying idea is that of a functional programming approach: all elements of the model state needed for a transition function will be given as arguments; the transition cannot be implicitly affected by any other mutable state or unintended side effects. While this may take some getting used to, it decouples the simulation code, eases the reasoning about the transition function and makes it easier to write unit tests for it.[4]

The fact that all transitions are defined on inputs from the previous step eases parallelization[5]. It also excludes artefacts that might arise from the order in which agents carry out their transitions.[6] Last but not least, it can be viewed as a representation of plans and conflicts that might arise between them: if, e.g., two agents want to take the same resource, or move to the same place in the case where only one may be in a place at each point in time, this means that methods to resolve the resulting conflicts are needed. The computational platform requires to make this explicit by one or the other mechanism, rather than having to assume "first-come-first-serve" as is usual with random order of transitions of agents. The system transition then recomposes the overall state from all recreated parts of the state, resolving potential conflicts as defined by the modeller.

Parallelization means that model simulation runs are computed on several processing elements (PE). Each of these executes the same program code, but for a different part of the data. When necessary, information is exchanged between the different PEs, using a standard like the message passing interface, MPI. To assign similar chunks of work to the different PEs, the graph is partitioned with the aim of balancing this load between them. Efficient graph partitioning is a well-researched topic in the HPC context. The platform does the parallelization mostly automatically, it is not necessary to keep parallelization in mind while writing transition functions.

Version 0 of the GCF computational platform for DT models is implemented in R, as it is intended to provide a grasp of how the platform functions and how it can be used. Future versions will focus on enhanced computational performance.

[4]Deterministic "random" generators then allow to transform a stochastic transition function to a deterministic one while testing the code.

[5]Otherwise, information sent from one process to another one might be outdated in the meantime

[6]This order is conventionally chosen to be random; in introductions to agent-based modelling, this convention is rarely questioned. Other ordering conventions are feasible.