Chapter 7
Agent-Based Modelling and Simulation with Domain-Specific Languages

Oliver Reinhardt, Tom Warnke, and Adelinde M. Uhrmacher

Conducting simulation studies within a model-based framework is a complex process, in which many different concerns must be considered. Central tasks include the specification of the simulation model, the execution of simulation runs, the conduction of systematic simulation experiments, and the management and documentation of the model's context. In this chapter, we look into how these concerns can be separated and handled by applying domain-specific languages (DSLs), that is, languages that are tailored to specific tasks in a specific application domain. We demonstrate and discuss the features of the approach by using the modelling language ML3, the experiment specification language SESSL, and PROV, a graph-based standard to describe the provenance information underlying the multi-stage process of model development.

7.1 Introduction

In sociological or demographic research, such as the study of migration, simulation studies are often initiated by some unusual phenomenon observed in the macro-level data. Its explanation is then sought at the micro-level, by probing hypotheses about decisions, actions, and interactions of individuals (Coleman, 1986; Billari, 2015). In this way, theories about decisions and behaviour of individuals, as well as data that are used as input, for calibration, or validation, contribute to the model generation process at the micro- and macro-level respectively. Many agent-based demographic simulation models follow this pattern, e.g., for fertility prediction (Diaz et al., 2011), partnership formation (Billari et al., 2007; Bijak et al., 2013), marriage markets (Zinn, 2012) as well as migration (Klabunde & Willekens, 2016; Klabunde et al., 2017). Whereas typically, data used for calibration and validation focuses on the macro-level, additional data that enter the model-generating process at micro-level add both to the credibility of the simulation model (see Chaps. 4 and 6) and to the complexity of the simulation study.

J. Bijak, *Towards Bayesian Model-Based Demography*, Methodos Series 17,
https://doi.org/10.1007/978-3-030-83039-7_7

An effective computational support of such simulation studies needs to consider various concerns. These include specifying the simulation model in a succinct, clear, and unambiguous way, its efficient execution, executing simulation experiments flexibly and in a replicable manner (see Chap. 10), and making the overall process of conducting a simulation study, including the various sources and the interplay of model refinement and of simulation experiment execution, explicit. Given the range of concerns, domain-specific languages (DSLs) seem particularly apt to play a central role within supporting simulation studies, as they are aimed at describing specific concerns within a specific domain (Fowler, 2010). In DSLs, abstractions and notations of the language are tailored to the specific concerns in the application domain, so as to allow the stakeholders to specify their particular concerns concisely, and others in an interdisciplinary team to understand these concerns more easily. The combination of different DSLs within a simulation study naturally caters for the separation of different concerns required for handling the art and science of conducting simulation studies effectively and efficiently (Zeigler & Sarjoughian, 2017).

In this chapter, we explore how different DSLs can contribute to (a) agent-based modelling (and present implications for the efficient execution of these models) based on the modelling language ML3, (b) specifying simulation experiments based on the simulation experiment specification language SESSL, and finally, (c) to relating the activities, theories, data, simulation experiment specifications, and simulation models by exploiting the provenance standard PROV. We also discuss a salient feature of DSLs, that is, that they constrain the possibilities of the users in order to gain more computational support, and the implication for use and reuse of the language and model.

7.2 Domain-Specific Languages for Modelling

DSLs for modelling are aimed at closing the gap between model documentation and model implementation, with the ultimate goal to conflate both in an executable documentation. Two desirable properties of a DSL for modelling are practical expressiveness, describing the ease of specifying a model in the language as well as how clearly more complex mechanisms can be expressed, and succinctness. Whereas the number of the used lines of code can serve as an indication for the latter, the former is difficult to measure. Practical expressiveness must not be confused with formal expressiveness, which measures *how many* models can theoretically be expressed in the language, or, in other words, the genericity of the language (Felleisen, 1991).

7.2.1 Requirements

A necessary prerequisite for achieving practical expressiveness is to identify central requirements of the application domain before developing or selecting the DSL. These key requirements related to agent-based models, specifically in the migration context, are listed below.

Objects of Interest. In migration modelling, the central objects of interest are the individual migrants and their behaviour. With an agent-based approach, migrants are put in the focus and represented as agents. In contrast to population-based modelling approaches, such an agent-based approach allows modelling of the heterogeneity among migrants. Each migrant agent has individual attribute values and an individual position in the social network of agents. As a consequence, agent-based approaches allow modelling of how the situation and knowledge of an individual migrant influences his or her behaviour. In addition to the migrant as the central entity, other types of actors can be modelled as agents in the system, for example government agencies or smugglers. Although these might correspond to higher-level entities, depicting them as agents facilitates modelling of the interaction between different key players in migration research.

Dynamic Networks. Agent-based migration models need to include the effects of agents' social ties on their decisions and vice versa. Therefore, both the local attributes of an agent and its network links to other agents should be explicitly represented in the modelling language. It is also crucial to allow for several independent networks between agents. This becomes particularly important when combining different agent types as suggested above, for example to distinguish contact networks among migrants from contacts between migrants and smugglers. Note that encoding changes in the networks can be challenging, both in the syntax of the DSL as well as in the simulator implementation.

Compositionality. Agent-based simulation models can become complex quickly due to many interconnected agents acting in parallel. All agents can act in ways that change their own state, the state of their neighbours, or network links. A DSL can address this complexity by supporting compositional modelling. As stated by Henzinger et al. (2011, p. 12), "[a] compositional language allows the modular description of a system by combining submodels that describe parts of the system". An agent-based model as described above can be decomposed into parts on several levels. First, different types of agents can be distinguished. Second, different types of behaviour of a single type of agent can be described independently. Both improve the readability of the model, as different parts of the model can be understood individually.

Decisions. A central goal of this simulation study is to deepen our understanding of migrants' decision processes (see Chaps. 3 and 6). Modelling these decisions in detail, and the migrants' knowledge on which they are based, is therefore inevitable.

The DSL must therefore be powerful and flexible enough to express them. In addition, the language must not be limited to a single model of decision making, to enable an implementation and comparison of different decision models.

Formal Semantics. Simulation models are often implemented in an *ad hoc* fashion. If a model is instead specified with a DSL and that DSL has a formal definition, it becomes possible to interpret the model or parts of it based on formal semantics. The semantics of a DSL for modelling maps a given model to a mathematical structure of some class, often a stochastic process. For example, many modelling approaches in computational biology are based on Continuous-Time Markov Chains (De Nicola et al., 2013). In addition to helping the interpretation of a model, establishing the connection between the DSL and the underlying stochastic process also informs the design of the simulation algorithm and, for example, allows reasoning over optimisations. Thus, DSLs for agent-based modelling of migration benefit from having a formal definition.

Continuous Time. In agent-based modelling, there are roughly two ways to consider the passing of time. The first approach is the so-called 'fixed-increment time advance,' where all agents have the opportunity to act on equidistant time points. Although that approach is the dominant one, it can cause problems that threaten the validity of the simulation results (Law, 2006, 72 *ff*). First, the precise timing of events is lost, which prohibits the analysis of the precise duration between events (Willekens, 2009). Second, events must be ordered for execution at a time point, which can introduce errors in the simulation. The alternative approach is called 'next-event time advance' and allows agents to act at any point on a continuous time scale. This approach is very rarely used in agent-based modelling, but can solve the problems above. Therefore, a DSL for agent-based modelling of migration should allow agents to act in continuous time.

7.2.2 The Modelling Language for Linked Lives (ML3)

Based on the above requirements we selected the Modelling Language for Linked Lives (ML3). ML3 is an external domain-specific modelling language for agent-based demographic models. In this context, *external* means that it is a new language independent of any other, as opposed to an *internal* DSL that is embedded in a host language and makes use of host language features. ML3 was designed to model life courses of interconnected individuals in continuous time, specifically with the modelling of migration decisions in mind (Warnke et al., 2017). That makes ML3 a natural candidate for application in this project. In the following Box 7.1, we give a short description of ML3, with examples taken from a version of the Routes and Rumours model introduced in Chap. 3, available at https://github.com/oreindt/routes-rumours-ml3, and relate it to the requirements formulated above.

Box 7.1: Description of the Routes and Rumours Model in ML3

Agents: The primary entities of ML3 models are agents. They represent all acting entities of the modelled system, including individual persons, but also higher-level actors, such as families, households, NGOs or governments. An agent's properties and behaviour are determined by their type. Any ML3 model begins with a definition of the existing agent types. The following defines an agent type `Migrant`, to represent the migrants in the Routes and Rumours model:

```
Migrant(
  capital : real,
  in_transit : bool,
  steps : int
)
```

Agents of the type `Migrant` have three attributes: their capital, which is a real number (defined by the type real after the colon), for example an amount in euro; and a Boolean attribute, that denotes if they are currently moving, or staying at one location; and the number of locations visited so far.

Agents can be created freely during the simulation. To remove them, they may be declared 'dead'. Dead agents do still exist, but no longer act on their own. They may, however, still influence the behaviour of agents who remain connected to them.

Links: Relationships between entities are modelled by links. Links, denoted by `<->`, are bidirectional connections between agents of either the same type (e.g., migrants forming a social network), or two different types (e.g., migrants residing at a location that is also modelled as an agent). They can represent one-to-one (`<->` e.g., two agents in a partnership), one-to-many (`<->` e.g., many migrants may be at any one location, but any migrant is only at one location), or many-to-many relations (`<->` e.g., every migrant can have multiple other migrant contacts, and may be contacted by multiple other migrants). The following defines the link between migrants and their current location in the Routes and Rumours model:

```
location:Location[1]<->[n]Migrant:migrants
```

This syntax can be read in two directions, mirroring the bidirectionality of links: from left to right, it says that any one `[1]` agent of the type `Location` may be linked to multiple `[n]` agents of the type `Migrant`, who are referred to as the location's `migrants`. From right to left, any `Migrant` agent is linked to one `Location`, which is called its `location`. ML3 always preserves the consistency of bidirectional links. When one direction is changed, the other is changed automatically. For example, when a new location is set for a migrant, it is automatically removed from the old location's migrants, and added to the new location's migrants.

(continued)

Box 7.1 (continued)

Function and procedures: The ability to define custom functions and procedures adds expressive power to ML3, allowing complex operations, and aiding readability and understandability by allowing for adding a layer of abstraction where necessary. Unlike many general-purpose programming languages, ML3 distinguishes functions, encapsulating calculations that return a result value, and procedures, containing operations that change the model state. Both are bound to a specific agent type, making them related to methods in object-oriented languages. A library of predefined functions and procedures aids with common operations. The following function calculates the cost of travel from the migrant's current location to a potential destination (given as a function parameter):

```
Migrant.move_cost(?destination : Location) : real :=
    costs_move * ego.location.link
    .filter(?destination in alter.endpoints).only().friction
```

The value of this function is calculated from the base cost of movement (the model parameter `costs_move`), scaled by the friction of the connection between the two locations, which is gained by filtering all outgoing ones using the predefined function filter, and then unwrapping the only element from the set of results using `only()`. The keyword `ego` refers to the agent the function is applied to. Procedures are defined similarly, with `->` -ing the:=.

Rules: Agents' behaviour is defined by rules. Every rule is associated with one agent type, so that different types of agents behave differently. Besides the agent type, any rule has three parts: a guard condition, that defines *who* acts, i.e., what state and environment an agent of that type must be in, to show this behaviour; a rate expression, that defines *when* they act; and the effect, that defines *what* they do. With this three-part formulation, ML3 rules are closely related to stochastic guarded commands (Henzinger et al. 2011). The following (slightly shortened) excerpt from the Routes and Rumours shows the rule that models how migrants begin their move from one location to the next:

```
1 Migrant
2 | !ego.in_transit                          // guard
3 @ ego.move_rate()                          // rate
4 -> ego.in_transit := true                  // effect
5    ego.destination := ego.decide_destination()
```

The rule applies to all living agents of the type `Migrant` (line 1). Like in a function or procedure, `ego` refers to one specific agent to which the rule is applied. According to the `guard` denoted by `|` (line 2) the rule applies to all

(continued)

Box 7.1 (continued)
migrants who are currently not in transit between locations. The `rate` following `@` (line 3) is given by a call to the function `move_rate`, where a rate is calculated depending on the agent's knowledge of potential destinations. The value of the rate expression is interpreted as the rate parameter of an exponential distribution that governs the waiting time until the effect is executed. Rules with certain non-exponential waiting times may be defined with special keywords (see Reinhardt et al., 2021). The effect is defined in lines 4 and 5, following `->`. The migrant decides on a destination and is now in transit to it.

In general, the guard and rate may be arbitrary expressions, and may make use of the agent's attributes, links (and attributes and links of linked agents as well), and function calls. The effect may be an arbitrary sequence of imperative commands, including assignments, conditions, loops, and procedure calls. The possibility of using arbitrary expressions and statements in the rules is included to give ML3 ample expressiveness to define complex behaviour and decision processes. The use of functions and procedures allows for encapsulating parts of these processes to keep rules concise, and therefore readable and maintainable.

For each type of agent, multiple rules can be defined to model different parts of their behaviour, and the behaviour of different types of agents is defined in separate rules. The complete model can therefore be composed from multiple sub-models covering different processes, each consisting of one or more rules. Formally, a set of ML3 rules defines a Generalised Semi-Markov Process (GSMP), or a Continuous-time Markov Chain (CTMC) if all of the rules use the default exponential rates. The resulting stochastic process was defined precisely in Reinhardt et al. (2021).

7.2.3 Discussion

Any domain-specific modelling language suggests (or even enforces), by the metaphors it applies and the functionality it offers, a certain style of model. Apart from the notion of linked agents, which is central for agent-based models, for ML3, the notion of behaviour modelled as a set of concurrent processes in continuous time is also of key importance. This is in stark contrast to commonly applied ABM frameworks such as NetLogo (Wilensky, 1999), Repast (North et al., 2013), or Mesa (Masad & Kazil, 2015), which are designed for modelling in a stepwise, discrete-time approach. If in a simulation model events shall occur in continuous time, these events need to be scheduled manually (Warnke et al., 2016). In this regard, and with its firm grounding in stochastic processes, ML3 is more closely related to stochastic process algebras, which have also been applied to agent-based systems before (Bortolussi et al., 2015). Most importantly, this approach results in a complete separation of the model itself, and its execution. ML3's rules describe these processes

declaratively, without including code to execute them (which we describe in the next section of this chapter). This makes the model more succinct, accessible and maintainable.

The result of applying ML3 to the Routes and Rumours model was twofold (Reinhardt et al., 2019). On the one hand, the central concepts of ML3 were well suited to the model, especially in separating the different kinds of behaviour into multiple concurrent processes for movement, information exchange, exploration and path planning. Compared to the earlier, step-wise version of the model (Hinsch & Bijak, 2019), this got rid of some arbitrary assumptions necessitated by the fixed time step, e.g., that movement to another location would always take one unit of time. In the continuous-time version, time of travel can depend on the distance and friction between the locations without restrictions.

On the other hand, it became apparent that some aspects of the model were difficult to express in ML3. In particular, ML3 knows only one kind of data structure: the set. This hindered modelling the migrants' knowledge about the world and the exchange of knowledge between migrant agents. These processes could be expressed, but only in a cumbersome way that, in addition, was highly inefficient for execution. The reason for this lack of expressive power is rooted in ML3's design as an external DSL, with a completely new syntax and semantics independent of any existing language. The inclusion of all the capabilities that general purpose languages have in regards to data structures would be possible, but would be unreasonable due to the necessary effort.

While the application of ML3 in this form was deemed impractical for the simulation model, insights from its application very much shaped the continued model development. The model was redesigned in terms of continuous processes, using the macro system of a general-purpose language (in this case, Julia) to achieve syntax similar to ML3's rules, as this excerpt, equivalent to the rule shown above, demonstrates:

```
1 @processes sim agent::Agent begin
2 ...
3 @poisson(move_rate(agent, sim.par))
4   ~ ! agent.in_transit
5   => start_move!(agent, sim.model.world, sim.par)
```

Line 1 is equivalent to line 1 in the ML3 rule (Box 7.1), with the difference that in ML3 the connection to an agent type is declared individually for every rule, while this version does it for a whole set of processes. Lines 3 to 5 contain the same three elements (guard, rate, effect) as ML3 rules, but with the order of the first two switched. The effect was put in a single function `start_move`, which contains code equivalent to that in the effect of the ML3 rule. This Julia version is, however, not completely able to separate the simulation logic from the model itself, but requires instructions in the effect, to trigger the rescheduling of events described in the next section.

In terms of language design, this endeavour showed the potential of redesigning ML3 as an internal DSL. ML3's syntax for expressions and effects already closely resembles object-oriented languages. Embedding it in an object-oriented host-language would allow the use of a similar syntax and other host-language features, such as complex data structures, type systems as well as tooling, for generating and debugging models.

7.3 Model Execution

When a simulation model is specified, it must be executed to produce results. If the model is implemented in a general-purpose language, this usually just means executing the model code. However, if specified in a DSL such as ML3, the model specification does not contain code for the execution, which is handled by a separate piece of software: the simulator. Given a model and an initial model state, i.e., a certain population of agents, the simulator must sample a trajectory of future states. For models with exponentially distributed waiting times, such as ML3, algorithms to generate such trajectories are well established, many of them derived from Gillespie's Stochastic Simulation Algorithm (SSA) (Gillespie, 1977). In the following, we describe a variation of the SSA for ML3. A more detailed and technical description can be found in Reinhardt and Uhrmacher (2017). The implementation in Java, the ML3 simulator, is available at https://git.informatik.uni-rostock.de/mosi/ml3.

7.3.1 Execution of ML3 Models

We begin the simulation with an initial population of agents, our state s, which is assumed at some point in time t (see Fig. 7.1a). As described in Sect. 7.2, each ML3 agent has a certain type, and for each type of agent there are a number of stochastic rules that describe their behavior. Each pair of a living agent a and a rule r matching the agent's type, where the rule's guard condition is fulfilled, yields a possible state transition (or *event*), given by the rule's effect applied to the agent. It is associated with a stochastic waiting time T until its occurrence, determined by an exponential distribution whose parameter is given by the rule's rate applied to the agent $\lambda_r(a, s)$. To advance the simulation we have to determine the event with the smallest waiting time Δt, execute its effect to get a new state s' and advance the time to the time of that event $t' = t + \Delta t$.

As per the semantics of the language, the waiting time T is exponentially distributed:

$$P\left(T \leq \Delta t\right) = 1 - e^{-\lambda_r(a,s)\cdot\Delta t}. \tag{7.1}$$

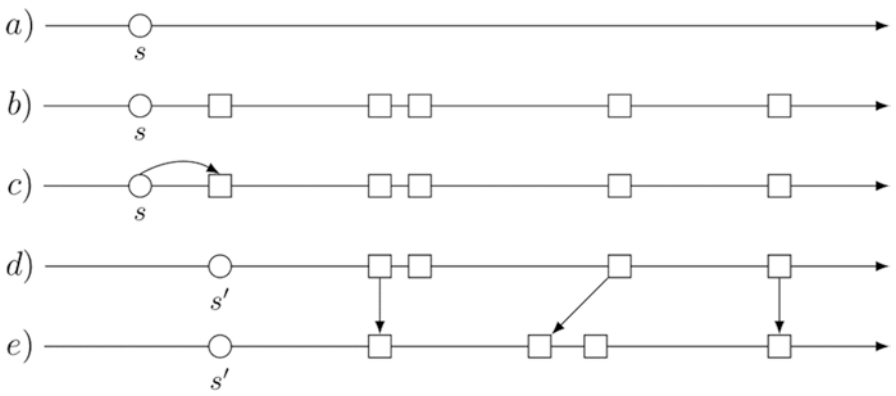

Fig. 7.1 Scheduling and rescheduling of events. We begin in state s at some time t depicted as the position on the horizontal time line (**a**). Events (squares) are scheduled (**b**). The earliest event is selected and executed (**c**), resulting in a new state s' at the time of that event (**d**). Then, affected events must be rescheduled (**e**)

This distribution can be efficiently sampled using inverse transform sampling (Devroye, 1986), i.e. by sampling a random number u from the uniform distribution on the unit interval and applying the distribution function's inverse:

$$\Delta t = -\frac{1}{\lambda_r(a,s)} \cdot \ln u \tag{7.2}$$

Using this method, we can sample a waiting time for every possible event (Fig. 7.1b). We can then select the first event, and execute it (Fig. 7.1c). In practice, the selection of the first event is implemented using a priority queue (also called the event queue), a data structure that stores pairs of objects (here: events) and priorities (here: times), and allows retrieval of the object with the highest priority very efficiently.

After the execution of this event, the system is in a new state s' at a new time t'. Further, we still have sampled execution times for all events, except the one that was executed (Fig. 7.1d). Unfortunately, in this changed state, these times might no longer be correct. Some events might no longer be possible at all (e.g., the event was the arrival of a migrant at their destination, so other events of this agent no longer apply). For others, the waiting time distribution might have changed. And some events might not have been possible in the old state, but are in the new (e.g., if a new migrant entered the system, new events will be added). In the worst case, the new state will require the re-sampling of all waiting times. In a typical agent-based model, however, the behaviour of any one agent will not directly affect the behaviour of many other agents. Their sampled times will still therefore be valid. Only those events that are affected will need to be re-sampled (Fig. 7.1e). In the ML3 simulator this is achieved using a dependency structure, which links events to attribute and link values of agents. When the waiting time is sampled, all used attributes and links are stored as dependencies of that event. After an event is executed, the

events dependent on the changed attributes and links can then be retrieved. A detailed and more technical description of this dependency structure can be found in Reinhardt and Uhrmacher (2017).

In Box 7.2 below, Algorithm 1 shows the algorithm described above in pseudo-code, and algorithm 2 shows the sampling of a waiting time for a single event.

Box 7.2: Examples of Pseudo-Code for Simulating and Scheduling Events

Algorithm 1 Simulation Algorithm.
s: the current state, given as a set of agents
t: the current time
m: the model, given as a set of rules
Q: the event queue
D: the dependency structure

```
// schedule all potential events in the event queue
for each a ∈ s, r ∈ m:
    if !dead(a, s) and typeof(a) = typeof(r):
        schedule(r, a)

while t < t_end:
    // select the next event from the queue
    (r, a, Δt) := pop(Q)

    // advance simulation time
    t := t + Δt

    // execute the event
    s := effect(r)(a, s)

    // reschedule the executed event
    schedule(r, a)

    // reschedule all affected events
    for each (r, a) ∈ affected(D):
        schedule(r, a)
```

Algorithm 2 Schedule.
(r, a): the event to schedule
s: the current state, given as a set of agents
t: the current time
m: the model, given as a set of rules
Q: the event queue
D: the dependency structure

```
if !dead(a, s) and guard(r)(a, s):
    u ~ Uniform(0,1)
    Δt := -1/rate(r)(a,s) · ln u
    push(Q,r,a,Δt)
else:
    remove(Q, r, a)
update(*D*)
```

7.3.2 Discussion

The simulation algorithm described above is abstract in the sense that it is independent of the concrete model. The model itself is only a parameter for the simulation algorithms – in the pseudo-code in Algorithm 1 in Box 7.2 it is called *m*. As a result, the simulator, i.e., the implementation of the simulation algorithm, is model-independent. All the execution logic can hence be reused for multiple models. This not only facilitates model development, it also makes it economical to put more effort into the simulator, as this effort benefits many models.

On the one hand, this effort can be put into quality assurance, resulting in better tested, more reliable software. A simulator that has been tested with many different models will generally be more trustworthy than an *ad hoc* implementation for a single model (Himmelspach & Uhrmacher, 2009). On the other hand, this effort can be put into advanced simulation techniques. One of these techniques we have already covered: using continuous time. The simulation logic for a discrete-time model is often just a simple loop, where the events of a single time step are processed in order, and time is advanced to the next step. The simulation algorithm described above is considerably more complex than that. But with the simulator being reusable, the additional effort is well invested. Separation of the modelling and the simulation concerns serves as an enabler for continuous-time simulation. Similarly, more efficient simulation algorithms, e.g., parallel or distributed simulators (Fujimoto, 2000), simulators that exploit code generation (Köster et al., 2020), or approximate the execution of discrete events (Gillespie, 2001) developed for the language, will benefit all simulation models defined in this language.

The latter leads us back to an important relationship between the expressiveness of the language and the feasibility and efficiency of its execution. The more expressive the modelling language, and the more freedom it gives to the modeller, the harder it is to execute models, and especially to do so efficiently. The approximation technique of Tau-leaping (Gillespie, 2001), for example, cannot simply be applied to ML3, as it requires the model state and state changes to be expressed as a vector, and state updates to be vector additions. ML3 states – networks of agents – cannot be easily represented that way. Ideally, every feature of the language is necessary for the model, so that implementing the model is possible, but execution is not unnecessarily inefficient. DSLs, being tailored to a specific class of models, may achieve this.

7.4 Domain-Specific Languages for Simulation Experiments

With the increasing availability of data and computational resources, simulation models become ever more complex. As a consequence, gaining insights into the macro- and micro-level behaviour of an agent-based model requires increasingly complex simulation experiments. Simulation experimentation benefits from using DSLs in several ways.

- They allow specifying experiments in a readable and succinct manner, which is an advantage over using general-purpose programming or scripting languages to implement experiments.
- They facilitate composing experiments from reusable building blocks, which makes applying sophisticated experimental methods to simulation models easier.
- They help to increase the trustworthiness of simulation results by making experiment packages available that allow other researchers to reproduce their results.

In this section, we illustrate these benefits by showing how SESSL, a DSL for simulation experiments, is applied for simulation experiments with ML3 and give a short overview of other current developments regarding DSLs for simulation experiments.

7.4.1 Basics

The fundamental idea behind using a DSL for specifying experiments is to provide a syntax that captures typical aspects of simulation experiment descriptions. Using this provided syntax, a simulation experiment can be described succinctly. This way, a DSL for experiment specification 'abstracts over' individual simulation experiments, by creating a general framework covering different specific cases. The commonalities of the experiments become then part of the DSL, and the actual experiment descriptions expressed in the DSL focus on the specifics of the individual experiments.

One experiment specification DSL is the 'Simulation Experiment Specification on a Scala Layer' (SESSL), an internal DSL that is embedded in the object-functional programming language Scala (Ewald & Uhrmacher, 2014). SESSL uses a more refined approach to abstracting over simulation experiments. Between the language core and the individual experiments, SESSL employs simulation-system-specific *bindings* that abstract over experiments with a specific simulation system. Whereas the language core contains general experiment aspects such as writing observed simulation output to files, the bindings package experiment aspects are tailored to a specific simulation approach, such as specifying which simulation outputs to observe. This way, SESSL can cater to the differences between, for example, conducting experiments with population-based and agent-based simulation models: whereas population-based models allow a direct observation of macro-level outputs, agent-based models might require aggregating over agents and agent attributes. Another difference is the specification of the initial model state, which, for an ML3 model, might include specifying how to construct a random network of links between agents.

To illustrate how experimentation with SESSL works, we now consider an example experiment specified with SESSL's binding for ML3 (Reinhardt et al., 2018). The following listing shows an excerpt of an experiment specification for the Routes

and Rumours model. Such an SESSL experiment specification is usually saved in a Scala file and can be run as a Scala script.

```
1 execute {
2   new Experiment with Observation {
3     model               = "routes.ml3"
4     replications        = 10
5     stopTime            = 100
6     set("p_find_links" <~ 0.5)
7     observeAt(stopTime)
8
9     initializeWith(JSON("init50.json"))
10    val migrants = observe("migrants" ~ agentCount(agentType = "Migrant"))
11    // additional lines elided
12  }
13 }
```

In an SESSL experiment, a number of options are available. For example, in the listing above, the model file, the number of replications, and the stop time of each simulation run are set in lines 3–5. Line 6 is an example of setting the value of a model input parameter, and line 7 specifies that model outputs are recorded when a simulation run terminates. These are examples of settings that are part of virtually all experiments and, therefore, belong to the SESSL core. The lines 9 and 10, in contrast, refer to settings that are ML3-specific and packaged in the SESSL binding for ML3. Line 9 specifies a JSON file that is used to create an initial population for each simulation run. An ML3-specific observable, which counts the number of `Migrant` agents, is configured in line 10.

Which options are available in an experiment depends on the binding used, but also the creation of the experiment as in line 2. Here, the experiment is configured to include observation options (**with** `Observation`). With such 'mix-ins,' SESSL allows a high degree of flexibility. Some mix-ins are packaged in the SESSL core and provide generic features; others belong to bindings and contain simulation-system-specific features. For example, the `Observation` mix-in above is part of the binding for ML3, and provides commands to record observations from ML3 simulation runs, such as `agentCount`.

This example shows how recurring aspects of simulation experiments can be efficiently expressed. Through bindings and mix-ins, SESSL allows for packaging code and making it available for reuse across experiments. As a result, the actual experiment specification focuses on the specifics of the experiment with little syntactical overhead.

7.4.2 Complex Experiments

The specification of more complex experiments in SESSL exploits the abstraction over different simulation systems. Many experimental methods can be integrated with the generalisation of simulation experiments in the SESSL core. As a result, those methods can be applied to any experiment for any simulation system. Examples of experimental methods that are realised this way are algorithms to create designs of experiments, which work with the inputs of an experiment (e.g., set in the experiment shown above), or algorithms that process the outputs.

We demonstrate this by fitting a regression meta-model to the Routes and Rumours model, based on a central composite design (see Reinhardt et al., 2018 for background). Based on the experiment specification shown above, three changes are necessary to integrate these experimental methods with the experiment. First, the mix-ins `CentralCompositeDesign` and `LinearRegression` are added to the experiment:

```
new Experiment with ... with CentralCompositeDesign with LinearRegression {
```

To the configuration options of the experiment we add the specification of the design.

```
centralComposite("p_drop_contact" <~ interval(0.0, 1.0), "p_info_mingle" <~ interval(
0.0, 1.0), ...)
```

Lastly, the linear regression is applied to the collected simulation results.

```
1 withExperimentResult { result =>
2   val regr = fitLinearModel(result)("p_drop_contact", "p_info_mingle", ...)(migrants)
3   println(regr.fittedFunction)
4   println(regr.rSquared)
5 }
```

This is an example of the extensibility of internal DSLs such as SESSL. The `withExperimentResult` block allows injecting arbitrary user code that is invoked when the experiment (all replications of all design points) is finished. Here, we use the function `fitLinearModel` to obtain a regression meta-model `regr` for the observed result, the given factors, and the observable migrants. The fitted function and the r^2 goodness-of-fit measure are written as output.

7.4.3 Reproducibility

In addition to making specifying and executing simulation experiments easier, DSLs can also help to make experiments reproducible (for a general discussion, see Chap. 10). As experiments are typically single files, they can be easily distributed to other researchers, who can then execute the experiments and confirm their results. This way, textual DSLs and, in particular, internal DSLs facilitate packaging experiments in an executable fashion, in contrast to, for example, GUI-based experimentation tools. However, the execution of an experiment requires additional software that must be acquired and installed. SESSL solves this challenge by employing Apache Maven (https://maven.apache.org/), an industry-grade software project management tool, and its associated infrastructure. We give a short summary of the idea below.

Each SESSL experiment is accompanied by a Maven configuration file (called `pom.xml`) that contains details about the software artefacts needed to execute the experiment. Those software artefacts might have their own dependencies, which are automatically resolved by Maven. For example, an SESSL experiment with an ML3 model must only declare its dependency on the SESSL binding for ML3, which in turn depends on the SESSL core and the ML3 simulation package. To execute an experiment, Maven checks whether all dependencies are already installed and, if not, downloads and installs all missing software artefacts automatically. Thus, these downloads are only necessary for the first execution of the experiment. An example of packaging an experiment this way is the SESSL-ML3 quickstart package, which is available from https://git.informatik.uni-rostock.de/mosi/sessl-ml3-quickstart.

7.4.4 Related Work

Using a tailored language to specify simulation experiments was pioneered by the 'Simulation Experiment Description Markup Language' (SED-ML) (Waltemath et al., 2011). SED-ML aims at computational biology and, being based on XML, is a machine-readable rather than human-readable language. In contrast to SESSL, where experiments are executable standalone artefacts, SED-ML is an exchange format for experiments that can be written and read by tools in the computational biology domain.

In the area of agent-based simulation, some tools support simple experiments. Repast Simphony, for example, provides an interface for 'Batch Runs,' which are simple parameter sweeps (Collier & Ozik, 2013); Netlogo's BehaviorSpace module (Wilensky, 2018) enables parameter sweeps as well. Both approaches allow importing and exporting experiments as XML files. In contrast to SED-ML, however, these XML files are tool-specific and cannot be used to port an experiment from one tool to another. More complex experiments can be implemented by writing code that generates such files. For example, this approach has been used to apply

Simulated Annealing (an optimisation algorithm) to a Repast Simphony model (Ozik et al., 2014). More recently, an R package with a DSL-like interface has been published that implements complex experiments by generating XML files for NetLogo (Salecker et al., 2019).

To gain more independence from concrete tools, simulation experiments can also be represented in a more abstract form, for example in *schemas* (Wilsdorf et al., 2019). Such a schema describes a machine-readable format of the salient aspects of a simulation experiment, which can then be used to (semi-) automatically generate representations of that experiment in concrete tool formats.

7.4.5 Discussion

Using DSLs emphasises the role of simulation experiments as standalone artefacts. Experiments and their parts can be composed and reused largely independently of a concrete simulation model, as they are defined in their own DSL. The DSL implementation is then responsible for executing a given experiment specification for a given model. In other words, DSLs for simulation experiments allow separation of the concerns of developing a model on the one hand, and designing experiments for a model on the other.

One central advantage of DSLs for simulation experiments is the potential for reuse. First, it becomes possible to reuse components of simulation experiments and compose new experiments from them. This is particularly useful when applying complex experimental methods to a simulation model, as these methods can be implemented based on an experiment abstraction that represents the commonalities of all simulation systems. By mapping a concrete simulation system to this abstraction, as SESSL's bindings do, all methods become applicable. But the term 'reuse' can also refer to complete experiments. One relevant example is conducting the same experiment with two different implementations of a model or two different models of the same phenomenon. By confirming that the results from both experiment executions match, the models can be *cross-validated*.

Finally, expressing simulation experiments with DSLs also facilitates capturing the role of experiments and their relation to simulation models in the course of a simulation study, which is studied in the following section by using the concept of formal provenance modelling.

7.5 Managing the Model's Context

Understanding how the data and theories have entered the model-generating process is central for assessing a simulation model, and the simulation results that are generated based on this simulation model. This understanding also plays a pivotal role in

the reuse of simulation models, as it provides valuable information as to for which applications a given model might be valid.

Documentation of agent-based models has been standardised in the ODD protocol (Overview, Design concepts, Details; see Grimm et al., 2006), which is regularly applied in many fields, including the social sciences (Grimm et al., 2020). However, ODD only includes small parts of the wider context, how a simulation model has been generated, mostly in the 'purpose' and 'input data' elements. Some more information (especially on analysis) is included by TRACE (Schmolke et al., 2010; Grimm et al., 2014), which, when applied to an agent-based model, might include an ODD documentation of the model itself. Both of these approaches rely on extensive textual descriptions, which might easily add up to 30 pages (see, e.g., Klabunde et al., 2015).

Instead of textual description, we propose a more formal approach, i.e., using PROV (Groth & Moreau, 2013), which represents a provenance standard, to describe how a simulation model has been generated (Ruscheinski & Uhrmacher, 2017). *Provenance* refers to "information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness" (Groth & Moreau, 2013).

PROV represents provenance information as a directed acyclic graph. This graph contains different types of nodes, including *entities* (shown as circles), e.g., data, theories, simulation model specifications, or simulation experiment specifications, and *activities* (shown as squares), such as calibration, validation, analysing, refining, or composing. Edges represent relationships between nodes, the most prominent ones being *used by* and *generated by*. For example, the entities simulation model and data may be used by the activity calibration, and as a result, a calibrated simulation model as well as an experiment specification be generated by this activity. DSLs do not need to be executable, and in fact PROV is not; however, it allows for storage of the information in a structured manner in a graph database and consequently, for it to be queried.

In this way, the analyst can query, for instance, which data have been used for validating or calibrating a particular model, or retrieve all validation experiments that have been executed with simulation models and upon which a particular simulation model is based. If DSLs, such as ML3, are used for specifying the simulation model, and other DSLs, such as SESSL, are used for specifying the simulation experiments, then these simulation experiments can be reused for future model versions (Peng et al., 2015) and may be re-executed automatically (Wilsdorf et al., 2020). Besides, provenance information can be stored and retrieved at different levels of detail (Ruscheinski et al., 2019). We illustrate this based on the Routes and Rumours model.

Figure 7.2 shows an example of a provenance graph, based on Box 5.1 in Chap. 5. It describes in detail how a sensitivity analysis was conducted. The provenance graph begins with the Routes and Rumours model, as defined in Chap. 3, on the very left (*M*). For the purpose of this example, we omit the process of the model creation, and the entities on which it is based. At first, as described in the second paragraph in Box 5.1, a Definitive Screening Design was applied on the 17 model parameters,

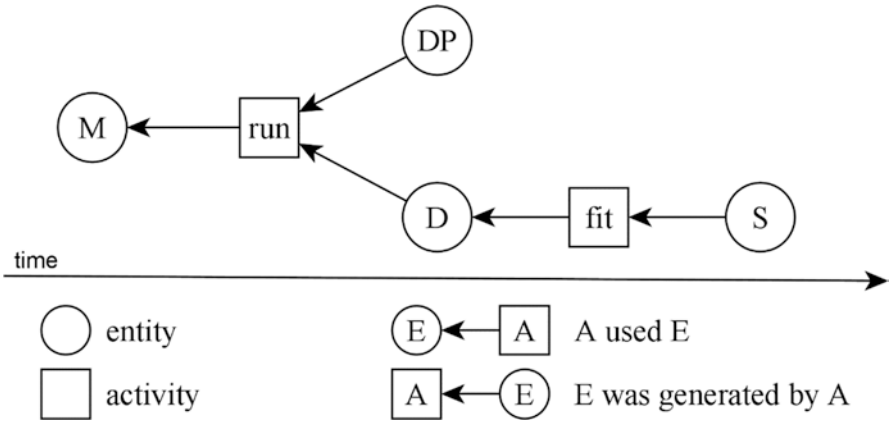

Fig. 7.2 Provenance graph for model analysis based on Box 5.1 in Chap. 5. (Source: own elaboration)

and simulation runs were performed on the 37 resulting design points. We model these two steps as a single process (*run*), which generated two entities: the design points (*DP*) produced in the design step, and the data produced by the simulation runs (*D*).

Subsequently, GP emulators were fitted to the data in the next step (*fit*), yielding the emulators and the information about sensitivity they contain (*S*) as a result. If this was conducted using a DSL such as SESSL (see Sect. 7.4), or even a general-purpose programming language, the processes (*run*) and (*fit*) would have yielded the corresponding code as additional products, which would appear as additional entities, and could be used to easily reproduce the results. However, the analysis was performed with GEM-SA, a purely GUI-based tool, so there is no script, or anything equivalent.

Figure 7.3 (see Appendix E for details) shows a broader view of the whole modelling process in less detail, including multiple iterations of models (*Mi*), their analysis, psychological experiments, and data assessment. The whole analysis shown in Fig. 7.2 is then folded into the process *a*1, the first step of the broader analysis of the Routes and Rumours model. The analysis shown above uses that model (*M*3) as an input, and produces sensitivity information as an output (*S*1). The process is additionally linked to the methodology proposed by Kennedy and O'Hagan (2001), denoted as (*K*01), and thereby indirectly related to the later steps of the process, in which a similar analysis is repeated on subsequent versions of the model.

To give the provenance graph meaning, appropriate information about the individual entities and activities must be provided. The type of entity or activity determines what information is necessary. That might be a textual description (e.g., ODD for models, or a verbal description of the processes as in Box 5.1), code (potentially in a domain-specific language), or the actual data and relevant meta-data for data-entities. In our case, to provide sources of this information, in Appendix E we mostly refer to the appropriate chapters and sections of this book.

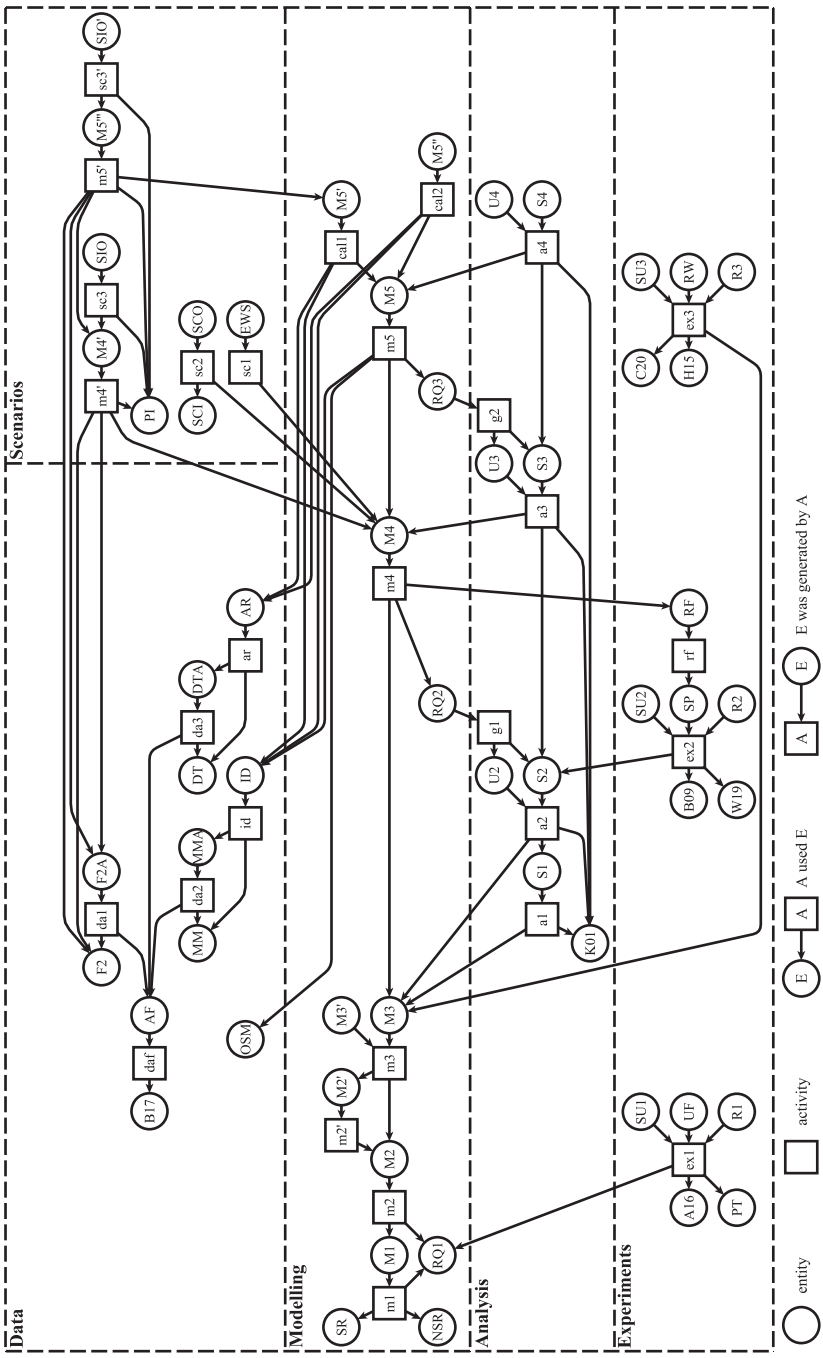

Fig. 7.3 Overview of the provenance of the model-building process – for details, see Appendix E. (Source: own elaboration)

Of course, as a natural extension, a provenance model may also span multiple simulation studies on related subjects, relating current research to previous research, for example if a model developed in one study reuses parts of a previous model (Budde et al., 2021). For this purpose, standardised provenance models included in model repositories such as CoMSES/OpenABM can be used.

7.6 Conclusion

Conducting a complex simulation study is an intricate task, in which a variety of different concerns have to be considered. We have identified some of the central ones, i.e., specifying a simulation model, executing simulation runs, conducting complex simulation experiments, and documenting the context and history of a simulation model, and demonstrated how domain-specific languages can be employed to tackle them separately. A domain-specific modelling language allows for a succinct model representation, making use of suitable metaphors. With the application of the ML3 to the Routes and Rumours model, we have demonstrated the value of such metaphors, e.g., ML3's rules to model concurrent processes. At the same time, DSLs put a limitation to the kinds of models that can be expressed. This limitation of expressive power, however, has benefits for the execution of simulation runs, in that limitations allow for more efficient simulation algorithms. A DSL that is too powerful for its purpose might hence be equally impractical. This highlights an important trade-off for selecting a suitable DSL – and for designing such a language in the first place. DSLs for simulation experiments allow the specification of such experiments in a readable and succinct way. Such executable experiment specifications may then be shared and reused, improving reproducibility of results.

Finally, PROV, a graph-based language for provenance modelling, allows the specification of a model's history and context in a way that is accessible to both human readers and computational processing. This is especially important for creating and documenting subsequent model versions as part of the iterative process advocated throughout this book, including several different elements, such as model versions, languages and formalisms used, empirical and experimental data, elements of analysis (meta-modelling and sensitivity) and their results, and so on. The creation of such model is presented in Chap. 8, and the role of individual elements in the whole model-building process, as well as its scientific and practical implications, are discussed throughout Part III of the book.