

## Chapter 6

# What Software Engineering has to offer to Agent-Based Social Simulation

Peer-Olaf Siebers and Franziska Klügl

### Abstract

In simulation projects it is generally beneficial to have a toolset that allows following a more formal approach to system analysis, model design, and model implementation. Such formal approaches are developed for supporting a systematic proceeding by making different steps explicit as well as by providing a precise language to express the results of those steps, documenting not just the final model but also intermediate steps. This chapter consists of two parts: The first gives an overview of which tools developed in Software Engineering can and have been adapted to agent-based social simulation; the second part demonstrates with the help of an informative example how some of these tools can be combined into an overall structured approach to model development.

### Why Read this Chapter?

To get to know the tools and techniques that Software Engineering has on offer when it comes to taking a more structured approach to model building. This is particularly useful for larger, collaborative and multi-disciplinary projects. Resulting models are easy to maintain and extend and are easy to communicate (and consequently to reproduce), even if the models themselves are highly complex.

**Keywords:** Agent Based Social Simulation, Software Engineering, Model Development, Formal Approach, Unified Modelling Language, Normative Comparison, Office Environment

## 6.1 Introduction

In most, if not all simulation projects, it is beneficial to proceed in a systematic way, even more for larger, collaborative and multi-disciplinary projects. Agent-based Social Simulation (ABSS) partially suffers from the fact that despite of its increasing popularity, there is no standard way of addressing model development, simulation handling etc. Many modellers are basically self-taught when it comes to processes and tools involved in designing and implementing an ABSS model. Developing an ABSS model is anything but a trivial endeavour given the conceptual depth, often unclear level of detail and complexities involved when handling software that contains more than one thread of control. Computer Science - in particular Software Engineering - has developed a set of tools that enables following a so called "formal" approach to system analysis, model design and implementation. Such elements of a systematic proceeding make different steps explicit as well as provide clear and precise languages to capture the concepts, content or assumptions of the model documenting not just the final result, but also intermediate steps. Such an approach is naturally used for model development if elements and processes of the targeted system are more or less accessible, empirically well embedded or assumptions to be take are clear. In the terminology of Boero and Squazzoni (2005) this refers to a type of model more towards the case-based model side of the spectrum of models.

Models on the other end - theoretical abstractions - are more associated with scientific endeavour of hypothesis building and testing. They consequently need a much more exploratory process. Nevertheless, scientific rigor anyway requires a systematic procedure producing reproducible results as also [Norling et al. \(2013\)](#) argue. Formal languages allow to clearly formulate what shall be contained in a

model which supports not just awareness in the overall study development process, but also facilitates more unambiguous communication between all involved partners, especially between the ones that implement the model and the rest of the group. Thus, exploratory modelling profits as formal approaches support the thoughtfulness of the modelling process. It helps to avoid model artefacts and supports sharing, reproducing and re-using the model. Eventually, the model is transformed into software and applying Software Engineering supports the development of well-structured, understandable software that is easy to maintain and easy to extend.

This chapter advertises formal tools for model conceptualization, software development and project management as offered by Software Engineering. In the context of Social Simulation these tools can either be used individually to help with specific modelling activities or to guide the entire modelling process. The chapter consists of two parts: In the first part of this book chapter we provide an overview of tools and techniques used in Software Engineering which have been used or suggested in a social simulation context. In the second part of this book chapter we demonstrate with the help of an informative example how some of these tools can be combined into a structured framework that allows a more formal approach to model development. The informative example is based on a real world study where we aimed with a multidisciplinary team to develop a simulation model to study different facets of normative comparison.

## **6.2 Review of Formal Approaches to Model Development**

Already in 2006, Richardini et al. identified a number of methodological problems supposed to hinder the wider adoption of agent-based modelling and simulation in the social sciences. In contrast to alternative forms of modelling and simulation, ABSS is assumed not to be suitable to follow shared, standardized conventions of how to proceed, how to describe or how to analyse a simulation model due to its exploratory, bottom-up nature aiming at reproducing emergent processes, etc. Building software in general had similar problems for a long time as the early (and ongoing) discussions on the nature of software development (art, engineering or craftsmanship) show (see for example Pyritz (2003)).

There are no underlying principles of physics or other established basic knowledge that could be used for building software in the same way as for example rules of statics for building bridges. Nevertheless, there was the need to systematize software development by developing guidelines, conventions and best practices that make the development process more engineering-like, producing software with intended quality in a predictable way. A number of process models have been invented with the Waterfall Model as the most well-known traditional approach or Extreme programming as a more modern, flexible compilation of best-practices. Characteristic of the former are a number of steps that express more and more detailed views onto the resulting software product while moving from a clarification of what needs to be eventually implemented, tested and maintained. In modern forms of software development approaches, fast prototyping and frequent testing are in the centre. Iterative development with code refactoring that improve software design, replaces clearly structured, systematic larger process steps by more or less organized smaller advance.

Although ABSS has particularities that preclude to take simulations models as just another kind of software, Software Engineering offers a large repertoire of languages and tools to support systematic and structured system analysis and development: Formal and structured text-based and diagram-based languages allow more precise formulations of model elements than natural language would do. By clarifying what needs to be formulated, those language not only guide model specification and documentation, but all phases in development. Specific process suggestions organize different views and model description elements into a sequence of steps that correspond to some best practice of how to proceed when designing, implementing and working with a simulation model. Those methodologies exist not only for simulation models in general, but also for agent-based simulations in particular. At some

stage in those processes, best practices on a more technical level support the translation of concepts into program code - such support is given by (software) pattern formalizing good solution to recurring problems.

In this section, we will different contributions of Software Engineering to support the development of ABSS models. We identified four areas of such formal instruments in the widest sense and organized the section according to these four elements: Methodologies as the first pillar suggest how to manage the overall development process in a structured and aware way, from formulating the objective behind the ABSS study to validating and deploying the simulation results. We hereby concentrate more on the elements of the overall process that relate to the phases from model conceptualization to software development as we think Software Engineering tools most relevant for those. The second pillar are structured and formal languages for expressing the concepts that are seen as relevant in the system under consideration. These languages can be used in the different steps for describing different views or elements of the model in an as unambiguous as possible way. In phases towards implementation of the model, pillar three and four become essential. Architectures and pattern form a way to capture best practices in model design and implementation, while tools support the implementation process directly.

### **6.2.1 First Pillar: Development Processes and Software Engineering Methodologies**

Social scientists seem to associate Software Engineering-based approaches with "formal systems" that enforce to apply a prescribed sequence of steps using formal languages far too rigorous to be appropriate for the mostly exploratory nature of simulation model building. Software Engineering is an engineering discipline that is concerned with all aspects of software production<sup>1</sup> In general, it defines a systematic process with steps that guide the developer from requirements elicitation to implementation, validation and sometimes even maintenance of the software.

Nowadays there is a variety of more or less formal approaches in Software Engineering together with some understanding which of the methods is suitable for which kind of problem. As coined in Sommerville (2016) for example games are usually developed by producing a sequence of prototypes while safety-critical software development appears to be highly formal with elaborated and analysed specifications. In the same way as there are very different applications of agent-based simulation, one may expect also very different ways of building agent-based simulation software. A formal process hereby shall ensure that the result possess particular qualities: The resulting software shall be reliable and trustworthy, produced in an economic way. While the latter may mean that the software is based on reusable components in a well-structured way, the former qualities for simulation software refer to reproducibility of results and validity of the implemented simulation model.

In general, software is usually not used by the programmer. Sommerville (2016) states that the most costs in a software project are costs of changing the software after it has gone into use. This can be also stated for simulation in general - not just when decisions have been taken supported by results of a simulation study or publications have been published presenting hypotheses and making statements based on the results of a simulation study. Discovering too late that the model contains artefacts or does actually not answer the question it should do, can be embarrassing in the best case, deadly in the worst.

#### **Generic Processes**

Simulation Engineering in general has setup a number of generic processes much on the abstraction level of general Software Engineering activities. Some process models are independent from the actual model

---

<sup>1</sup> Sommerville (2016) relates Software Engineering also to Computer Science. The latter focusing more on theory and fundamentals, while Software Engineering is more practically oriented towards developing and delivering useful software. He also sees Software Engineering as a part of Systems Engineering which aims at systems integrating hardware, software and process engineering.

paradigm. Basically every simulation textbook proposes a procedure that basically organizes activities such as done in Law (2007), Shannon (1998), and Robinson (2004) or the stages of simulation-based research in Gilbert and Troitzsch (2005). Figure 1 shows a generic version of this process. This model development cycle starts with an explicit statement of the objective that is behind the simulation study undertaken, i. e. a formulation of the problem actually addressed. In a second step, the system is analysed, that means its basic components and their relations are determined. Reliable information and data sources are to be found for informing the different steps in the model development process. Based on this analysis, a conceptual model is specified elaborating the structure of the model, as well as the dynamics of all the interacting elements.

The conceptual model is hereby particularly important, as it helps not just to understand the system under consideration, but also to guide the subsequent phases by documenting the hypothesis taken. In the second part of this chapter we will elaborate an example approach to developing such a conceptual model. It is quite common – not just in our example – to elaborate the conceptual model from highly abstract descriptions of model elements and different points of view onto the model to a more and more concrete specification that can directly inform implementation.

Depending on the tools used for implementation, the production of a runnable and thus simulateable representation of the model is achieved as output of the next phase. Also this phase usually happens in an iterative way, either by adding more and more details to the model implementation or by fast prototyping and adapting. In the last phase, the model is deployed and experimented with for producing the intended results that are then documented and used. Each of the model representations produced in the different phases must sufficiently correspond to the original system (Validation), this is ensured by testing the models individually and by verifying that one representation is sufficiently related to the other. Figure 2 (presented in Section 6.3.1) is elaborating this process towards ABSS model development. The focus of this chapter is put on these earlier steps of a full study, only indicating how implementation can be achieved and widely omitting running experiments and analysing produced data.

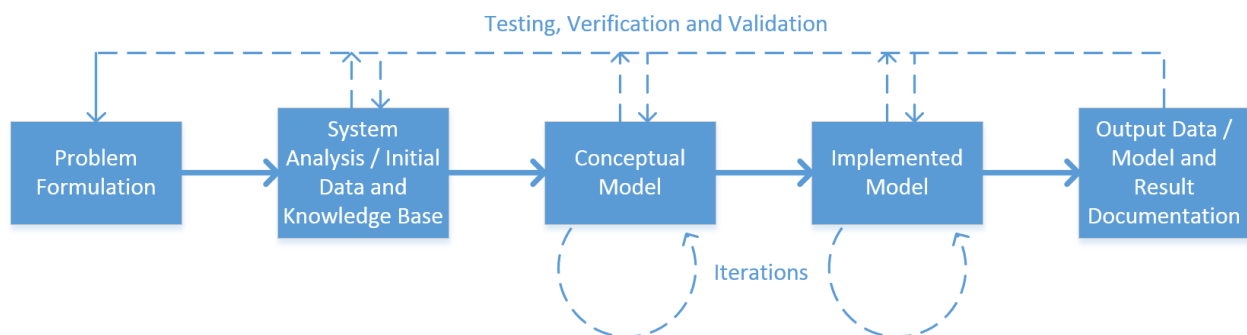


Figure 1: Generic steps in a simulation study

One can also find similar suggestions for structured procedure when developing a simulation study in ABSS. An example is Drogoul et al. (2003). They give more specific detail about types of knowledge and roles of different human experts involved. Activities are more detailed with respect to domain model (real agents), design model (conceptual agents) and operational model (computational agents).

Using the similarity of tools and languages used for modelling the conceptual views on software systems consisting of multiple agents and system analysis and model development in ABSS, there are a number of suggestions to extend methodologies developed for Agent-Oriented Software Engineering (AOSE). One can see AOSE as an extension of Object-oriented Software Engineering addressing the specific problems that arise when developing multi-agent systems, for an overview of different methodologies see Bergenti et al. (2004) and Gomez-Sanz and Fuentes-Fernandez (2015). Winikoff and

Padgham (2013) give a good introduction into the general principles of AOSE. One of the earliest AOSE methodologies that has been used to develop agent-based simulations was INGENIAS (Gomez-Sanz et al. 2010).

### **Specific Processes for ABSS**

Also detailed, formal methodologies have been proposed that are specific for developing ABSS models. Two examples proposing approaches similarly structured like AOSE methodologies are easyABM (Garro and Russo 2010) and MAIA (Ghorbani et al. 2013).

easyABM assumes different phases from System Analysis, Conceptual System Development, Simulation Design and Code Generation to Simulation Setup, Execution and Results Analysis. Particularly elaborated is the Conceptual System Modelling phase consisting of the development of different partial models that capture relevant views onto the model. An overall meta-model is provided for the different aspects that provides clear high-level language concepts and their relations. The Structural System Model contains sub-models for each component, determining its abstraction level. The main components are Society (composed), Agents (active), Artifacts (passive, resource manager). An interaction model describes how intra- and inter-relationships end in interactions. Hereby, a society model describes a society based on its composition, type and rules (safety rules and liveness rules). The central aspect of the Agent model is a complex goal model. It also contains a behaviour model composed of activities for achieving these goals, as well as interactions with other agents and artifacts. The latter are specified using behaviour and interactions. UML class and activity diagrams form the main mean to express those partial models. Developing the conceptual model further, simulation design is given in a language that resembles elements of the Repast Symphony meta model and thus enables at least partial code generation. The case study used in Garro and Russo (2010) to exemplify the usage of easyABM is a logistics scenario using simulation to test different management policies for vehicles stacking and moving containers. easyABM is already characterized as a model-driven approach and further developed towards MDA4ABMS (see below).

MAIA also focusses on conceptual modelling activities, yet social and society aspects play a specifically central role capturing social phenomena. It guides modelling institutions and social constructs based on a meta-model derived from the Institutional Analysis and Development framework of Ostrom (2005) already used in several agent-based simulation studies. The basic assumption is that social rules and institutions are more easily accessible to modellers than capturing individual behaviour. The MAIA meta-model is organized in five sub-models resembling different aspects of the underlying framework (Ghorbani et al. 2013): 1) Collective Structure with actors and their attributes, 2) Constitutional Structure with roles, their dependencies and actions, institutional statements such as norms, shared strategies, etc. 3) Physical Structure, 4) Operational Structure focussing on system dynamics and 5) Evaluation Structure containing concepts to evaluate and validate the outcomes of the system. In Ghorbani et al. (2014), these structures were extended by formally grounded operational semantics. This makes the specification given using the MAIA meta-model executable so that a runnable simulation can be directly generated it.

A purer methodology focussing on interactions is IODA (Kubera et al. 2011). The starting point of this methodology is the identification of interactions that simulated reactive agents exhibit with other agents as well as their simulated environment.

### **Model-Driven Development**

The basic idea of Model-Driven Development is that software development may consist of handling models of the intended software starting from a generic level (Stahl et al. 2006). Specifications then can be (semi-) automatically transformed into more and more platform-specific representations, eventually generating code. Basically one can see this approach as the currently most formally grounded, controlled

evolution of software. Adapting this idea to ABSS means that based on a precise formulation of the conceptual model subsequent, more and more concrete models are elaborated until finally a version that is fully adapted to a particular simulation platform is achieved. The above described specific methodologies for developing ABSS models, MAIA and easyABM, can be seen as first steps towards model-driven development methodologies. Garro et al. (2013) introduce MDA4ABMS as a complete model-driven approach proposing clearly defined meta-models for each of the major phases of development. There are ABSS specific meta-models on different levels of abstraction starting from a Computation Independent Model (CIM) on a conceptual level, a Platform Independent Models (PIM) with more specific architectural and behavioural details to a Platform-Specific Model (PSM) towards realization for a specific software platform. MDA4ABMS gives also guidelines and rules for the transition between the different phases of development – making even partially automatic transformation possible. The process is exemplified with an extended prisoner dilemma model.

Such methodologies clearly define what elements a system analysis needs to contain - underlying meta-models create a particular awareness behind the conceptualization. The assumption is that - if the original system is analysed sufficiently thoroughly and the results of this analysis written down in a sufficiently clear way - the simulation model can be communicated and implemented without uncertainties. The critical activity is developing a conceptual model. The formal elements of the methodologies shall sharpen the way how the modeller looks onto the system and guide overall model formulation in a reliable way even, for models in which the individual agents exhibit complex behaviour. Model-Driven Development works best in combination with Domain Specific Languages (DSL) that provide abstractions specific for a given application domain. Beyond taking an ABSS specific language as a DSL, there are not so many more specific languages yet. The meta-models mentioned above actually provide DSLs for ABSS with a particular perspective in mind. MAIA puts its focus more on institutions, easyABM more on the complex goal-directed behaviour of individual agents. Franchi (2012) proposed a specific language for agent-based social network modelling. Scherer et al. (2015) describe a model-driven approach for conceptual modelling phases specific for the public policy domain. Their toolset supports a semi-automated transformation of conceptual model representations to formal policy models and then to executable simulations of different scenarios. Their conceptual model is systematically derived from narrative texts. The conceptual model representation at the centre of their approach is specific for public policy development process. This adaptation to the policy domain makes the overall process particularly suitable for involving different stakeholder groups.

### **Agile Approaches**

Such structured methodologies seem to resemble more classical waterfall-type of Software Engineering approaches. Knublauch (2002) reports experiences with using Extreme Programming as a more modern, agile approach to developing agent-based software. Extreme Programming (Beck 2004) is more like a collection of best practices and principles such as "on-site customer" resulting in daily contacts to stakeholders for avoiding that the system develops into something which is actually not intended. Another principle is "Simple Design", that means producing software that solves the particularly addressed problem and nothing else. With "Refactoring", it is ensured that the quality of software design is improved after each iteration in the development cycle. "Short releases" as a principle means many executable prototypes and software testing are in the centre of the methodology. These principles are as important as the often more remarked "Pair programming" way of implementation - in which two persons sit in front of the monitor programming together - one coding, the other supporting. Extreme programming as overall approach may fit also to developing model specifications and simulation system formalizations using structured methodologies mentioned above. Short releases and testing would then correspond to running and analysing prototypic simulation runs in an overall iterative approach.

Moyo et al. (2015) organize the development of an agent-based simulation study using SCRUM, an agile approach to manage software development. This article forms a good introduction to agile software development methodologies for simulation in general and gives a case study modelling alcohol consumption dynamics.

### **Formal Methodologies versus Modelling Principles**

None of these more formal methodologies for developing agent-based simulations actually contradicts to the principles or informal strategies that are proposed in the social simulation community for model development. Examples for those principles are the KISS principle stating that a model should be as simple as possible. A yet not simpler principle is the KIDS strategy (Edmonds and Moss 2004) arguing that a model should be preferred that is understandable and descriptive. Simplification should not be exaggerated, especially before fully understanding the system to be modelled. Another strategy is the so-called pattern-oriented modelling (Grimm et al. 2005) that focuses on reproducing all pattern or stylized facts observable in the underlying data. General guidelines from a simulation engineering point of view can be found in Kasaie and Kelton (2015), but also in Richiardi et al. (2006). All these informal strategies can be combined with the more formal methodologies mentioned here. Underlying meta-models are usually very generic and can be used to capture many different societies, agents, etc.

### **6.2.2 Second Pillar: Structured and Formal Languages**

In contrast to natural language, structured and formal languages offer a mean to clearly describe a system. Formal languages form important elements of the methodologies discussed in the last section, but have also a value on their own. Syntax and semantics of language elements and their relations are precisely given. They may be so precise that a model fully described in a formal language may even be automatically processed - execution or analysis may be done without running the description. Often formal languages are distinct from programming languages due to their higher abstraction level enabling more meaningful constructs based on a clearly defined meta-model. Due to this high-level property, descriptions in the formal language can be more compact and focussed on the relevant aspects. Consequently, they are apt for specification and documentation. The clearly defined, underlying meta-model may be at first sight more restrictive than natural language, but the positive side of the restriction may be that it may result in more precise and clearer description.

Some of the later described languages are embedded into frameworks for being executable. That means, it may be possible to directly run a simulation specified in that language without translation into a programming language; if this is not fully possible, there might be a chance to create a code skeleton from the description that can be complemented for a full implementation. Even without any implementation, specification in some formal languages can be processed directly for deriving properties or for comparing the formulated model with likewise formalized high-level system descriptions.

There is a plethora of formal languages that can be used for capturing ABSS models or their elements. Different languages have different foci and are useful for different objectives, or as Edmonds (2004) puts it: "Formal Systems (such as logics) are not the *content* of theory but merely a *tool* for expressing and applying theory in a symbolic way." (p.1, italics in the original). So, they form an instrument for expressing a model or elements of a model. The first group of languages that may come into one's mind when thinking about formal languages are logic-based. Many different logical languages exist each of them focuses on a particular elements or uses a different starting point (Fasli 2004).

### **Logic-based Languages**

Languages for logic-based modelling correspond to mathematics as a language for analytical modelling. The language comes with limits resulting in that particular details cannot be formulated. If those details are not relevant when modelling a system, such a formal language is preferable as using the language

makes tools available for fast or even automated analysis, for fast simulation, etc. An example for a useful tool for ABSS based on logics is the LEADSTO language (Bosse et al. 2005). Its statements as extension of predicate logics can be used for expressing time dependencies between statements: T. Bosse suggests to use that logic for describing the overall dynamics of a simulation model so that it be automatically tested in the output data whether the statements hold in the actual simulation runs. In AOSE logic-based languages play in particular a role in the area of verifiable specification languages (for a review see Mascardi et al. (2004)).

### **Algebraic Specification Languages**

Besides formal logics, there are many languages that can be used for describing agent-based software as well as ABSS models. D'Inverno and Luck (2001) for example used the algebraic specification language Z for formally describing different multi-agent systems and their features for clarifying and finally understanding the core concepts. There are a lot of approaches for formalizing particular aspects, such as architectures, organizations, etc. Examples that are relevant as they not just cover agents in isolation are Weyns and Holvoet (2004) and Helleboogh et al. (2007). The language used there are more or less formal algebraic, but less structured than for example Z. However, the contents are particularly interesting as they show how interactions between agents and between agents and their environment can be captured in a precise and unambiguous way. They also demonstrate on what level of detail a fully clear specification would need to be given.

### **Petrinets**

Another example of a formal language that has been used for specifying multi-agent systems on different levels of aggregation are Petrinets. Köhler et al. (2007) show how social theories can be formalized using this graphical language of "places" and "transitions" with "tokens" traveling through the network. A place may hold token, while a transition transports token from one place to another on a strictly local basis. In computer science, Petrinets form an established modeling tools for concurrent, interacting processes and their synchronization. They are amendable for theoretical analysis, but their overall state changes - when becoming too complex for analysis - can also be simulated as places and transitions have a clearly given semantics. For expressing agent interaction and behavior, complex token structures are need to actually represent a network on their own.

### **Object-Oriented Simulation and DEVS**

In the object-oriented simulation community formal specification languages have been invented and found wide dissemination. The most prominent example is DEVS (Discrete EVent Specification) initially introduced by Zeigler (1990) is a specification language for object-oriented simulation models that is based notions from general system science. Initially, restricted to discrete event modelling and simulation, meanwhile it is seen as a more general approach that also can be used for continuous systems. An atomic model hereby consists of a description of the input, state and output variables, a description of which value combinations of input variables are fed into the running simulation ("Input segment"), transition functions for updating state and output variables, as well as a time advance function that characterizes how time is updated. Atomic models can be aggregated to composed models. Due to its generality, DEVS was advertised for using in ABSS by Duboz et al. (2006). Hocaoglu et al. (2002) focus on the giving more structure to the state of an atomic model enabling more complex agent behaviour. Specifications formulated in DEVS can be executed using specialized environments such as JAMES (Himmelspach et al. 2010).

### **Object-Oriented Software Specification and UML**



In AOSE the most prominent language for specifying particular views onto the overall system is UML (Unified Modeling Language). UML was developed for supporting Software Engineering processes (from requirement analysis to implementation and documentation) by providing a language consisting of different specialized diagrams that address different aspects of an object-oriented software system (Fowler 2003). It is actually a semi-formal diagram language. That means, it allows some extent of vagueness when describing a system. There is an additional formal language - OCL, the object constraint language - that can be used to add information that cannot be expressed in the diagrams directly.

The first edition of UML - used for developing object-oriented software - was defined in the mid of the 90ies as an integration of different diagram notation in different object-oriented modelling systems. Especially in AOSE, there have been a number of suggestions for extensions, e.g. class diagrams containing information about the particular social organization; behaviour diagrams containing structures for particular agent architectures, etc. Those extensions were mostly done for specific AOSE methodologies. The most known extension for software agents was AgentUML (Odell et al. 2000) which mostly concerned the sequence diagrams for enabling the formulation of more flexible and diverse interactions and reactions to messages than simple method calls. Some of these extensions became part of the UML standard 2, in a graphically different way than proposed by AgentUML; as now alternatives and conditional reactions can be formulated in UML 2, AgentUML has been declared to be obsolete (Bauer and Odell 2005).

As ABSS in general are often designed and implemented using object-oriented languages and tools, Bommel and Müller (2007) motivate the usage of UML diagrams as a suitable tool for communication between different experts involved in a simulation project. A good introduction to UML for ABSS can be found in (Bersini 2012) or (Siebers and Onggo 2014).

UML proposes various types of diagrams to capture different aspects of an overall object-oriented software system. The following diagrams types are mostly used in an agent-based simulation/system context. In the second part of this chapter we will illustrate their usage in more detail.

- **Use case diagrams** show different scenarios how the user may interact with the system. One could use it not only for users but also - on a coarse level - for interactions between an agent and its environment
- **Class diagrams** show the static structure of the software system by connecting classes to more general ones, or showing which classes are composed of others or classes that are otherwise linked to each other. This diagram is not just suitable for depicting the internal setup of an agent, but also its embedding into an organization structure.
- **State and Activity diagrams** can be used to capture dynamics. They show the states that a (typical) entity can be in, as well as their transitions. Activity diagrams focus on behaviour as a flow of activities also in relation to other agents' activities
- **Sequence diagrams** show how entities interact as a sequence of messages that they exchange.

In addition, other diagrams types are proposed to capture details of the package structure, deployment, etc. In the second part of this chapter, we give more details on how to use UML diagrams for model development and embed their usage into some form of best-practice process guiding the development of a conceptual model.

### 6.2.3. Third Pillar: Architectures and Patterns

Methodologies and the usage of a formal and precise language to describe different aspects of the conceptual model as well as to capture model specification, etc. are particularly important for people just starting with using modelling and simulation as they provide guidelines for managing the development process and support for conceiving a model. A third pillar of Software Engineering for ABSS is related to best practices in designing models that means they provide advice how to structure and build the actual software.

In the seminal book on Software Pattern (Gamma et al. 1994), such best practices in (object-oriented) software design have been formalized and captured in a way that they can be communicated and even taught. Over the years software design patterns have been suggested for many problem types, each of them giving a particular abstract "good" solution. Patterns have been also suggested in AOSE (see Juziuk et al. (2014) for a general survey). As North and Macal (2011) state, the standard software patterns are only of limited use for ABSS as the problems to be solved by them are of a completely different nature than to be solved when developing simulations. For simulation, one may distinguish two different views on design patterns: 1) design patterns that directly concern how to model particular phenomena or 2) design patterns that solve problems on a more technical level. Klügl and Karlsson (2009) give two examples for the first type of pattern, e.g. they describe what agent behaviour can produce exponential agent number growth. North and Macal (2011) give a list of patterns for the second case, e.g. patterns for agent scheduling, how to design spatial environments in an efficient way or the model-view-controller pattern which is also the most well-known pattern in Software Engineering describing how to separate visualization from application-specific logic.

Agent architectures can be seen as specific patterns for agent-based systems. Depending on whether human decision making shall be reproduced in a way that resembles how humans think or whether the agents need to exhibit complex and flexible behaviour, different architectures can be used. For the former type, so-called cognitive architectures such as SOAR (Laird et al. 1987; Wray and Jones 2005) or ACT-R (Anderson et al. 2004; Taatgen et al. 2006) have been suggested (for a short overview see Jones (2005)). Those architectures resemble theories from Cognitive Science supported by results from experiments with humans. Especially SOAR has been used for reproducing human behaviour in military training systems (Wray et al. 2005).

Although often indicated, the so-called BDI architecture is not a cognitive agent architecture, but a practical reasoning architecture (Wooldridge 2009). Its underlying motivation consists of a human-inspired means-end analysis separating the decision about which goal ("Desire") to pursue from the actual planning towards the goal the agent is committed to achieve ("Intention"). The BDI architecture has turned out to be very useful for software agents in general. It also appears to be a reasonable choice for organizing the internal decision making of agents in simulation, especially when more sophisticated agent behaviour needs to be formulated (see for example Joo (2013), Caillou et al. (2015), or Norling (2003)). Even in simulations with rather simple agent behaviour, it is advisable to use an agent architecture to organize the behaviour description, so that the agent program is more transparent, better readable and thus better analysable and maintainable.

Although not introduced as agent architectures, the general setup of rule-based systems, state automata or decision trees can provide important ways to structure agent behaviour descriptions and separate agents' decision making from the actual processing. A rule-based system contains a set of rules as "if... then..." constructs and a mechanism that systematically tests the current perception and agent state against the if-parts of the constructs. If this is true, the second part, the "then..." part, is activated. Using such a setup instead of a cascade of if-then-else programming language statements supports clarity of design and extensibility of the decision making model. Similar are decision trees which form another way avoiding ugly, inflexible implementations with hard-wired if-then-else cascades. A decision tree is a data structure that organizes conditions in nodes and different alternative values for those conditions in the branches out of the node. Another architecture pattern are state automata with an explicit representation of the state that the agent is in. The state is associated with particular behaviour. State changes happen based on a trigger relevant in the current state. An older, slightly more complex agent architecture following those ideas is the EMF frame (Drogoul and Ferber 1994). All agent architectures presented in AOSE can also be seen as local patterns for developing agents. They suggest a structure that supports to design and implement agents with non-trivial behaviour programs. Clearly, those architectures can be useful for ABSS as well.

In addition to software design patterns and agent architectures, there is another category of (software) pattern relevant for ABSS. These are meta-pattern capturing best practices in working with a model, not directly related to the model design or to a specific methodology. A good example is the ODD protocol (Grimm et al. 2013) for documenting ABSS models. It describes a framework of elements that make up a complete and useful documentation. One can also interpret any description of best practices for model testing, validation, etc. as such a meta-pattern. Rossiter (2015) describes a reference architecture for a simulation system in general clearly structuring the overall software into different layers of functionality. He also uses this reference architecture to explain the setup of existing platforms and to introduce a new toolkit.

#### 6.2.4 Fourth Pillar: Tools and Development Environments

There are many useful tools available for all phases of developing and using ABSS models. For the purpose of this chapter, we want to single out two particular types. Specialized drawing tools and software development platforms.

The diagrams capturing a model in for example UML may become quite large and complex. Thus tools that offer specialized shapes and other convenient support such as grid-based layout alignment, automated connections, etc. are highly valuable for making the drawing process more efficient and enable the modeller to concentrate on the important aspects of the description. Especially for UML, there are a number of good tools available, such as Visual Paradigm<sup>2</sup> or Visio<sup>3</sup>. Some platforms for implementing ABSS models, as for example Repast (Ozik et al. 2015) or Anylogic (see below), come with tools for drawing some UML diagrams types that are then directly translated into code skeletons.

Professional software development is usually done using an Integrated Development Environment (IDE). This is basically a collection of tools facilitating software development, such as elaborated program editors with build-in syntax checks, code completion, etc. allowing the programmer to concentrate on the semantics of the program rather than its syntax. Prominent IDE examples are Visual Studio<sup>4</sup> or Eclipse<sup>5</sup>. Such development environments also support for example code documentation by providing tools that automatically generate UML class diagrams from source code.

Inspired by those general IDEs and in addition to low-level programming support, an IDE for ABSS could contain

- Conceptual views on the implemented model with diagrammatic representations of what happens in the model. Drawing tools can be integrated with automated code generation from diagrams representing agent and organizational structures and agent behaviour and interaction dynamics.
- Simulation runtime support - tools for handling simulated time and space (maps), animation, inspection tools for individual agents and their interactions
- Appropriate ways to integrate model documentation, e.g. facilities to add comments or specific elements of an ODD model documentation.
- Automated generation of simulation runs including interfaces for conducting elaborated tests or manipulating model settings during runtime
- Debugging and validation support
- Convenient tools for defining experiments and input and output data handling

Such tools make model handling more convenient and efficient, yet they are built around a particular simulation platform that manages and executes a particular model implementation.

---

<sup>2</sup> [www.visual-paradigm.com](http://www.visual-paradigm.com) A free for non-commercial use community version exists

<sup>3</sup> [products.office.com/en/visio/](http://products.office.com/en/visio/)

<sup>4</sup> [www.visualstudio.com/](http://www.visualstudio.com/)

<sup>5</sup> [eclipse.org/](http://eclipse.org/)

Various specialized platforms for ABSS are available that aim at giving specific support. Over the last decades, hundreds of platforms and tools have been suggested. A Wikipedia page<sup>6</sup> lists 89 tools (in April 2016). Wikipedia also provides an up to date list of their attributes. Only a few of them deserve to be called an IDE for ABSS such as Repast Symphony ([repast.github.io/repast\\_simphony.html](http://repast.github.io/repast_simphony.html)), AnyLogic ([www.anylogic.com/](http://www.anylogic.com/)) or SeSAm ([www.simsesam.org](http://www.simsesam.org)). In addition to that list, there are a number of partially outdated surveys (Nikolai and Madey 2008; Railsback and Lytinen 2006; Kravari and Bassiliades 2015). The most prominent platforms are NetLogo ([ccl.northwestern.edu/netlogo/](http://ccl.northwestern.edu/netlogo/)) and Repast ([repast.sourceforge.net/](http://repast.sourceforge.net/)) and are part of each of the surveys. Other analysed platforms include AnyLogic, MASON ([cs.gmu.edu/~eclab/projects/mason/](http://cs.gmu.edu/~eclab/projects/mason/)), Gama ([gama-platform.org](http://gama-platform.org)) or Swarm (<http://www.swarm.org>).

Which platform to use is depending on a variety of factors ranging from the modellers personal preferences and experience to the properties of the model to be implemented. Also whether the platform is a commercial one or open source often plays a role. A general advice about the "best" platform is impossible.

### 6.3 Illustrative Example: Normative Comparison in an Office Environment

Up to now we have seen that Software Engineering in general and AOSE in particular offers a lot of support for developing ABSS models. Most of this support can be coined "formal": at the heart are clearly given process models describing the different steps to go through when doing a simulation study. This is particularly important for less experienced modellers as these process models help to solve the problem of translating vague mental representations of models into descriptions that are more and more refined. These methodologies help to know where one should start when doing a simulation study.

In the following we show based on an illustrative example that there is no need to be afraid of formal approaches, but that they can indeed be useful to support awareness about the actual model content when developing a model.

#### 6.3.1 Our Structured Approach

When aiming to develop ABSS models one faces the question of how to build them and where to start. This can be challenging not only for novices in the field but also for multidisciplinary teams where it is often difficult to engage everyone in the modelling process. Over the years we have developed a quite sophisticated "plan of attack" in form of a framework that guides the model development and can be used by either individuals or teams.

When used by individuals, they need to consider the perspective of potential team members (i.e. slip into their roles) during each process step. When used by teams, co-creation is an important aspect. Team members need to be open minded about the use of new tools and methods and about the collaboration with researchers from other domains and business partners. This is often not easy either for researchers trained in more traditional approaches or for business partners who often expect researchers to act like consultants, providing them with a report and a list of recommendations (Mitleton-Kelly 2003).

The framework supports model reproducibility through rigorous documentation of the conceptual ideas, underlying assumptions and the actual model content. The framework provides a step-by-step guide to conceptualising and designing ABSS models with the support of Software Engineering tools and techniques. Figure 2 provides an overview of the steps that make up the development process.

---

<sup>6</sup> [https://en.wikipedia.org/wiki/Comparison\\_of\\_agent-based\\_modeling\\_software](https://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software), accessed 07/05/2016

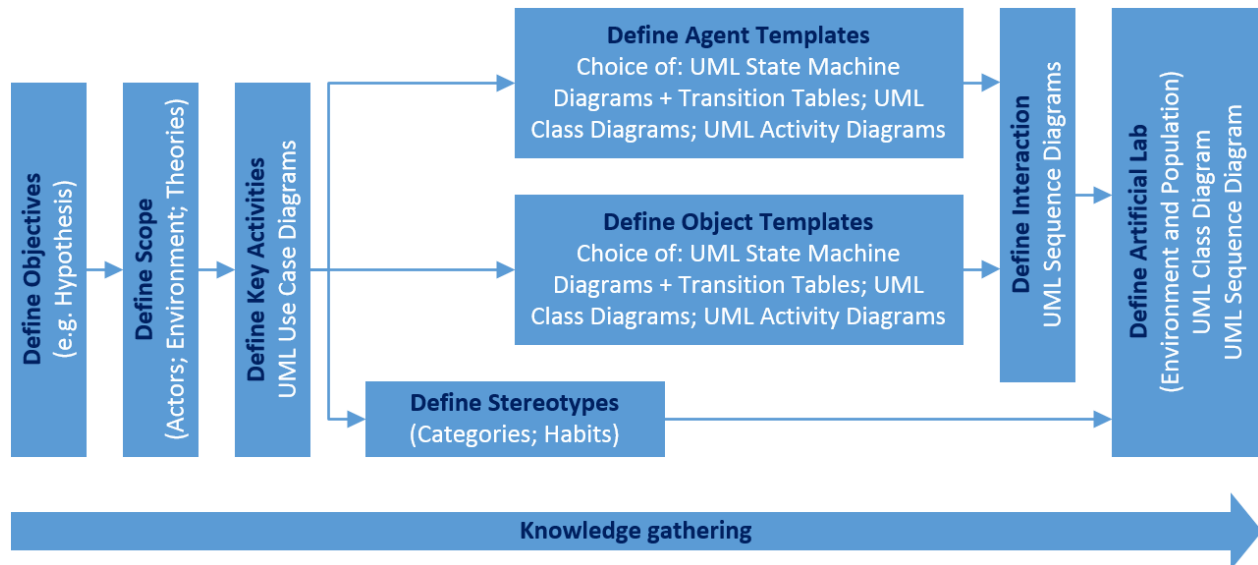


Figure 2: Overview of our ABSS development framework

While the framework represents a structured modelling approach, there will always be iterations required by the users to improve definitions from previous tasks. When stepping through the framework the users may realise that they did not consider important elements/details in a previous step or that they considered too many or that they considered them wrongly. In particular *discussions in focus groups* unearth these kinds of issues and are therefore extremely valuable for the model development process. The framework is suitable tool for well organised discussions and to capture the knowledge and ideas coming out of these discussions in a formal way. While there is a given sequence of steps that users should follow they need to be prepared to go back to a previous task if required and apply changes. Consequently this means that the users do not have to worry too much if in the initial rounds they get things wrong or things feel incomplete. They should simply move on to the next task if they feel that they have some form of contribution. Our experience is that it is necessary to revisit each task 4-5 times before there is a satisfying result that is acceptable for all stakeholders. In that sense, the approach somehow resembles "agile" approaches of Software Engineering with frequent interactions with stakeholders and frequent iterations, and not investing a lot of time into specifications that are obsolete after the next discussion.

While this framework will not work perfectly for all possible cases, it provides at least some form of systematic approach. The user should be prepared to adapt it to fit individual needs. In the following we will explain each step (including the necessary tools) and exemplify its application.

In order to demonstrate the use of our structured approach we use an illustrative example which is based on work by Zhang et al. (2011), Susanty (2015), and Bedwell et al. (2014). In this example we focus on the simulation model development to support studying the impact of normative comparison amongst colleagues with regards to energy consumption in an office environment. Normative comparison in this context means giving people clear regular personalised insight into their own energy consumption (e.g. "you used x% more energy than usual for this month") and allowing them to compare it to that of their neighbours (e.g. "you used x% more than your efficient neighbours"). A simulation study could compare the impact of "individual apportionment" vs. "group apportionment" of energy consumption information on the actual energy consumption within the office environment.

### 6.3.2 Gathering Knowledge

The task of knowledge gathering is one that happens throughout the structured modelling approach and in many different ways. The main ones we use in our framework are *literature review*, *focus group discussions*, *observations*, and *surveys*. The knowledge gathering is either a prerequisite for tasks (e.g. a literature review) or is embedded within the tasks (e.g. focus group discussions). For our study all focus groups were led by a Computer Scientist (the initiator of the study) and the participants consisted of a mixture of academics and researchers from the fields of Computer Science, Business Management, and Psychology. In this example study we did not engage with business partners. The team consisted of five core members that would participate regularly in the focus groups. Over the years we have made the experience that for our purposes smaller focus groups work best. Whenever we describe a task in the following we also briefly mentioned when and how the required knowledge was gathered.

### 6.3.3 Defining the Objectives

The first step within the framework is to define objectives in relation to the aim of the study. In our case this was done through a combination of a *literature review* and *focus group discussions*. After some iteration we came up with the following:

- Our aim is to study normative comparison in an office environment
- Our objective is to answer the following questions:
  - What are the effects of having the community influencing the individual?
  - What is the extent of impact (significant or not)?
  - Can we optimise it using certain interventions?
- Our hypotheses are:
  - Peer pressure leads to greener behaviour
  - Peer pressure has a positive effect on energy saving

With the objectives defined we then need to think about how we can test these objectives. For this we need to consider relevant experimental factors and responses. Experimental factors are the means by which it is proposed that the modelling objectives are to be achieved. Responses are the measures used to identify whether the objectives have been achieved and to identify potential reasons for failure to meet the objectives (Robinson 2004). In other words, experimental factors are simulation inputs that need to be set initially to test different scenarios related to the objectives while responses are simulation outputs that provide insight and show to what level the objectives have been achieved. In our case the hypotheses are very helpful for defining an initial set of experimental factors and responses:

- Experimental factors
  - Initial population composition (categorised by greenness of behaviour)
  - Level of peer pressure ("individual apportionment" vs. "group apportionment")
- Responses
  - Actual population composition (capturing changes in greenness of behaviour)
  - Energy consumption (of individuals and at average)

The experimental factors and responses defined at this stage are still very broad and need to be revisited when more information about the model becomes available.

### 6.3.4 Defining the Scope

At this stage we are interested in specifying the model scope. This requires some initial knowledge gathering. We did this through a *literature review* and *observation* of the existing system. With the help of the knowledge gathered we were then able to define the scope of the model. Decisions were made through *focus group discussions*. To guide the discussion and to document the decisions made in a more formal way we used an adaptation of the conceptual modelling *scope table* proposed by Robinson (2004) specially tailored towards ABSS modelling. The general categories we consider are: "Actor", "Physical Environment", and "Social / Psychological Aspect".

In order to make decisions about including or excluding different elements within these categories we asked ourselves, amongst others, the following questions:

- What is the appropriate level of abstraction for the objective(s) stated before?
  - This would define the level of abstraction acceptable.
- Do the elements have a relevant impact on the overall dynamics of the system?
  - Then they should be included.
- Do the elements show similar behaviour to other elements?
  - Then they should be grouped.

After some *discussions within the focus group* we decided that "transparency" would be the key driver for our decision making and that we want to abstract/simplify as much as possible while still keeping a realistic model (i.e. we aimed to explicitly follow the KISS principle mentioned in Section 6.2.1). In order to have easy access to data we decided to use our own offices (University of Nottingham; School of Computer Science) as the data source. Table 1 presents the resulting scope table in which we state for every element whether we want to include or exclude it and why we decided either way.

Category	Element	Decision	Justification	
Actor	Staff	Include as group (User)	Regularly occupy the office building	
	Research fellows			
	PhD students			
	UG+MSc students	Exclude	Do not have control over their work environment	
	Visitors	Exclude	Insignificant energy use	
Physical Environment	Appliance	HVAC (Heating + Ventilation + Aircon) system	Exclude	We only need one major energy consumer to test the theory; we decided to go for electricity
		Lighting	Include	Interacts with users on a daily basis; controlled by user
		Computer	Include	Interacts with users on a daily basis; controlled by user
		Monitor	Exclude	Modelled as part of the computer
		Continuously running appliances	Exclude	Constant consumption of electricity; not controllable by individuals
		Personal appliances	Exclude	No way to measure consumption
	Weather	Temperature	Exclude	Not necessary for proof-of-principle
		Natural light level	Exclude	Not necessary for proof-of-principle
	Room	Office	Include	Location where electronic appliances are installed
		Lab	Exclude	Mainly used by UG+MSc
		Kitchen	Include as group (Other Room)	Common areas frequently used by "users"
		Toilet		
		Corridor	Include	Commonly used when "users" move around
Social / Psychological Aspect	Comparative feedback	Include	Effective strategy to reduce energy consumption in residential building	
	Informative feedback	Include	Effective strategy to remove barriers in performing specific behaviour	
	Apportionment level	Include	Potential strategy to reduce energy consumption in office building	
	Freeriding	Include	Behaviour that differentiate two apportionment strategy	
	Sanction	Include	Factor to encounter freeriding behaviour	
	Anonymity	Include	Factor to encounter freeriding behaviour	

Table 1: Scope table for our illustrative example

### 6.3.5. Defining Key Activities

Interaction can take place between actors and between an actor and the physical environment it is in. Capturing these at a high level can be done with the help of *UML use case diagrams*. In Software Engineering *UML use case diagrams* are used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors). These diagrams do not make any attempt to represent the order or number of times that the systems actions and sub-actions should be executed. The relevant components of a use case diagram are depicted and described in Table 2.





Component	Symbol	Description
Actors		Entities that interface with the system (this can be people or other systems). Think of actors by considering the roles they play.
Use cases		Denotes what the actor wants your system to do for them.
System boundary		Indicates the scope of your system: The use cases inside the rectangle represent the functionality that you intend to implement.
Relationships		There are different types of relationships. In a relationship between use case and actor the associations indicate which actors initiate which use cases. A relationship between two use cases specifies common functionality and simplifies use case flows. We use <<Include>> when multiple use cases share a piece of same functionality which is placed in a separate use case rather than documented in every use case that needs it. We use <<Extend>> when activities might be performed as part of another activity but are not mandatory for a use case to run successfully. We are adding more capability.

Table 2: Relevant use case diagram components

While in Software Engineering the actors are outside the system boundaries (they are usually the users of software, and the software represents the system) when using use case diagrams in an ABSS context the actors are inside the system (representing the humans that interact with each other and the environment). The system boundaries are the boundaries of the relevant locations (which in our case would be the building boundaries of the office environment). It is important to understand that the purpose of these diagrams is to promote understanding; as long as they capture the ideas and help to explain them they are very useful. The use case diagram which we developed for our illustrative example through *focus group discussions* is depicted in Figure 3.



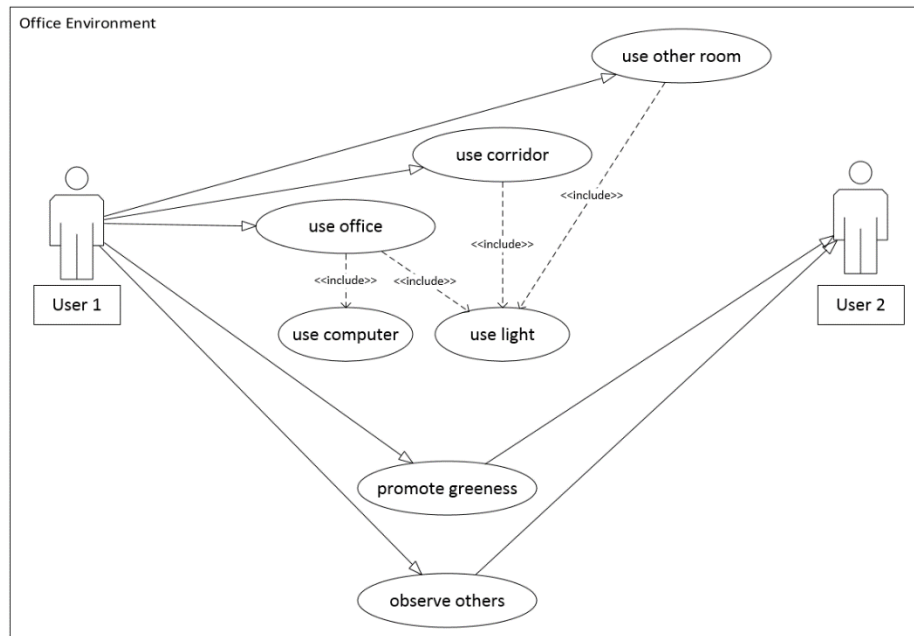


Figure 3: Use case diagram for our illustrative example [drawn with Visio]

### 6.3.6 Defining Stereotypes

In social psychology, a stereotype is a thought (or belief) that can be adopted about specific types of individuals or certain ways of doing things (McGarty et al. 2002). In order to be able to represent a specific population in our simulation models we define stereotypes that allow us to classify the members of this population. We derived our stereotype templates (categories, habits to be considered, and type names) through *focus group discussions* and through considering the *knowledge gathered previously*. Getting the stereotype templates right is more an art than a science. After long debates we decided to have two categories of stereotypes: one related to "work time", the other related to "energy saving awareness". Once the categories were identified we had to come up with the habits that describe these stereotypes:

- Habits for work time category
  - Arrival time at office
  - Leaving time from office
- Habits for Energy Saving Awareness category
  - Energy saving awareness
  - Likelihood of switching off unused electric appliances
  - Likelihood of promoting greenness

To get the information we needed to fully define the stereotypes we conducted a *survey* amongst our school's academics, researchers, and PhD students, anonymously asking them questions about their habits towards work time and energy saving awareness. We then *analysed the data* through cluster analysis to come up with the stereotype groups, assigned some speaking name and populated the stereotype tables with the "habit" information. The stereotype definitions we ended up with can be found in Table 3 and 4.

Stereotype	Working days	Arrival time	Leave time
Early bird	Mon-Fri	5am-9am	4pm-7pm
Time table complier	Mon-Fri	9am-10am	5pm-6pm
Flexible worker	Mon-Fri	10am-1pm	5pm-11pm
Hardcore worker	Mon-Fri + Sat	8am-10am	5pm-11pm

Table 3: User stereotypes defining work time habits

Stereotype	Energy saving awareness [0-100]	Probability of switching off unnecessary appliances	Probability of sending emails about energy issues to others
Environmental champion	95-100	0.95	0.9
Energy saver	70-94	0.7	0.6
Regular user	30-69	0.4	0.2
Big user	0-29	0.2	0.05

Table 4: User stereotypes defining energy saving habits

### 6.3.7 Defining Agent and Object Templates

For each of the relevant actor types we have identified in our scope table we have to develop an agent template containing all information for a prototypical agent. These templates will act as a blueprint when we later create the actor population for each simulation run. When it comes to modelling the environment we need similar templates for everything relevant we have identified in the scope table that lends itself to be represented as an object (e.g. the appliances). For other things (e.g. the weather) we need to consider other modelling methods. From a technical point of view there is no big difference between agents and objects. Thus we can use the same types of diagrams to document their design. We will therefore use the term "entity" when we talk about both. There are three different diagram types that are of relevance for defining entity templates: UML class diagrams (to define structure), UML state machine diagrams (to define behaviour), and UML activity diagrams (to define logic). Often only a subset of these is required. When developing the templates we create the different diagrams in parallel and in an iterative manner as often one informs and inspires the development of the other. As with the stereotypes, getting the entity templates right is not hard science and will therefore require many iterations.

In Software Engineering *UML class diagrams* are used to define the static structure of the software to be developed by showing classes (which are blueprints to build specific types of objects) and the relationships between classes. These relationships define the logical connections between classes (association, aggregation, composition, generalisation, dependency). UML class diagrams can be very complex and for our purposes it is often enough to consider individual classes. Therefore we focus on how to define individual classes here. In UML classes are depicted as rectangles with three compartments. The first compartment is reserved for the class name. This is simply the name of the entity as defined in the scope table (e.g. "User" for our user agent template). The second compartment is reserved for attributes (constants and variables). Often we would capture key state variables (e.g. "energy saving awareness"), key parameters and key output variables (e.g. "own energy consumption") here. The third compartment is reserved for operations that the user may perform. For each operation, we define some function names that indicate what kind of additional code we have to produce later (e.g. "moveToNewLocation()"). The brackets indicate that this is a function. Figure 4 shows as an example the user class definition we developed in parallel with the other template diagrams in several *focus group discussion* sessions.

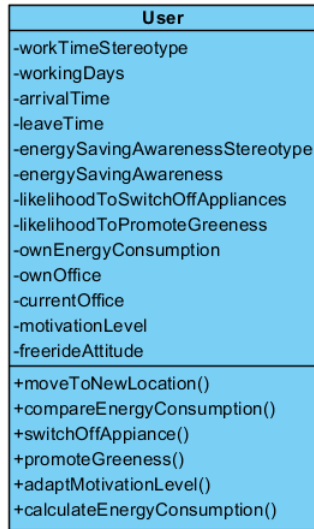


Figure 4: User class definition [drawn with Visual Paradigm]

In Software Engineering *UML state machine diagrams* (sometimes just called "state chart") are used to represent the dependencies between the state of an object and its reaction to messages or other events. State machine diagrams show the states of a single object, the events or the messages that cause a transition from one state to another and the action that result from a state change. A state machine diagram has exactly one state machine entry pointer which indicates the initial state of the agent. A state in a state machine diagram models a situation during which some invariant condition holds. Usually time is consumed while an object is in a specific state. A simple state is a state that does not have sub-states while a composite state is a state that has sub-states (nested states). The relevant components of a state machine diagram are depicted and described in Table 5.









Component	Symbol	Description
Entry pointer		Indicates the initial state after an object is created
State		Represents a locus of control with a particular set of reactions to conditions and/or events
Initial state pointer		Points to the initial state within a composite state
Final state		Termination point of a state chart
Transition		Movement between states, triggered by a specific event
Branch		Transition branching and/or connection point
Shallow history		The state chart remembers the most recent active sub state (but not the lower level sub-states)
Deep history		The state chart remembers the most recent active sub state (including the lower level sub states)

Table 5: Relevant state machine diagram components

In our case we use state machine diagrams to define the behaviour of our entities. This type of diagram is particular useful as it can be automatically translated into source code by IDEs who support such features. One can use several diagrams (e.g. one representing physical states and one representing mental states) for the same entity. A state machine diagram is not always meaningful (e.g. if there are no relevant states that need to be represented to capture the behaviour) or necessary (e.g. "energy saving awareness" could be expressed in states "aware" and "not aware" but also as a state variable that represents the level of awareness). There is nothing wrong with having entity templates without state

machine diagrams. While for Software Engineering the descriptions of how transitions are triggered are usually embedded within the diagram (in a rather cryptic language) it might be a good idea to present them in a separate table, to make the diagram easier to understand.

Many people find it difficult to get started with developing the state machine diagrams for agent templates. In order to come up with potential states that an agent can be in it helps to think in terms of *locations* (e.g. "in office"). The next step would be to think about *key time consuming activities* within these locations (e.g. "working with computer"). It is important to consider only key locations and key activities as otherwise the state chart gets too complex. One should only define as much detail as is really necessary for investigating the question studied. The above steps are just suggestions and do not always work. In case they do not work one has to use intuition and try to draft something that "feels right".

Figure 5 shows as an example the user state machine diagram we developed in parallel with the other template diagrams in several *focus group discussions*. Here we have defined location states based on the relevant rooms we identified in the scope table and added one location ("outOfOffice") to represent the outside world. The ideas for the activity states stem from our use case diagram (Figure 3). We then added transition arrows to represent the possible transitions between the defined states. Transitions with a question mark symbol are condition triggered while transitions with a clock symbol are time triggered. If there are more than one transition connecting states we have considered different triggers for state changes. This becomes clearer when we look at the transition definitions in Table 6. Here we can see that for example a state change from "outOfOffice" to "inCorridor" can happen for all user stereotypes during the working week and only for hard-core worker user stereotypes during the week end.

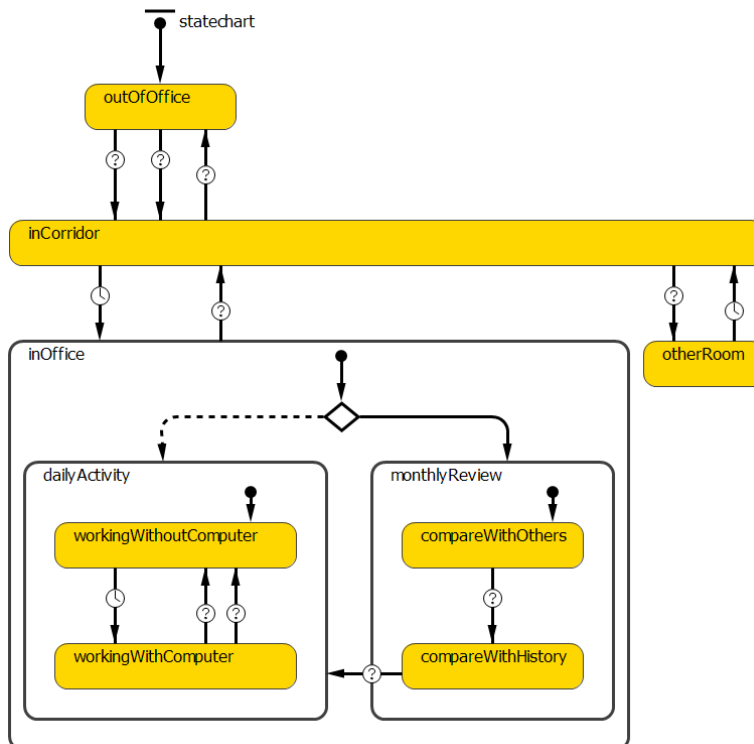


Figure 5: User state machine diagram [drawn with AnyLogic]

From state	To state	Triggered by	When?
outOfOffice	inCorridor	Condition	At typical arrival time during the working week for all
outOfOffice	inCorridor	Condition	At typical arrival time on Saturdays for hard-core workers only
inCorridor	outOfOffice	Condition	At typical leave time
inCorridor	inOffice	Timeout	At average after 5 minutes
inOffice	inCorridor	Condition	At random while at work or when leaving
inCorridor	otherRoom	Condition	At random while at work
otherRoom	inCorridor	Timeout	At average after 10 minutes
...	...	...	...

Table 6: User state machine transition definitions (excerpt)

In Software Engineering *UML activity diagrams* describe how activities are co-ordinated (the overall flow of control). They represent workflows of stepwise activities (while state machine diagrams show the dynamic behaviour of an object) and actions with support for choice, iteration and concurrency. Often people describe activity diagrams as just being fancy flow charts. The relevant components of an activity diagram are listed in Table 7.



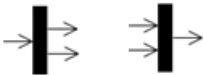



Component	Symbol	Description
Activity		Named box with rounded corners (a state that is left once the activity is finished)
Activity edge		Arrow (fires when the previous activity completes)
Synchronisation bar		Represent the start (split) or end (join) of concurrent activities
Decision diamond		Used to show decisions
Start marker		Indicate entry point of the diagram
Stop marker		Indicate exit point of the diagram

Table 7: Relevant activity diagram components

Amongst others, we can use these activity diagrams as a formal way to describe a decision making process (logic flow). In our case we use it to describe the logic flow of the normative comparison process. In order to define the logic flow we use the information we gathered from our *literature review* on psychological factors in the scoping phase. Figure 6 shows as an example the actions happening when the user agent is in the state "compareWithHistory" (which in the model is triggered once per simulated month). It is good practice to provide some evidence from the literature for the rationale behind the decision making process. This would come from our scoping phase *literature review* but might also require some additional resources. As an example, let's take the case "Less than former month? = no / Group? = yes / Sanction? = yes / Not Anonymous?". In the literature we find that using mechanisms to identify freerides and implement sanctions (social (e.g. gossip) or institutional (e.g. fines)) reduces the likelihood of further free-riding (Fehr et al. 2002). This is our justification for adding the action "decrease freeriding" for this case. In the end we would evaluate our logic flow by *discussing it in the focus group*.

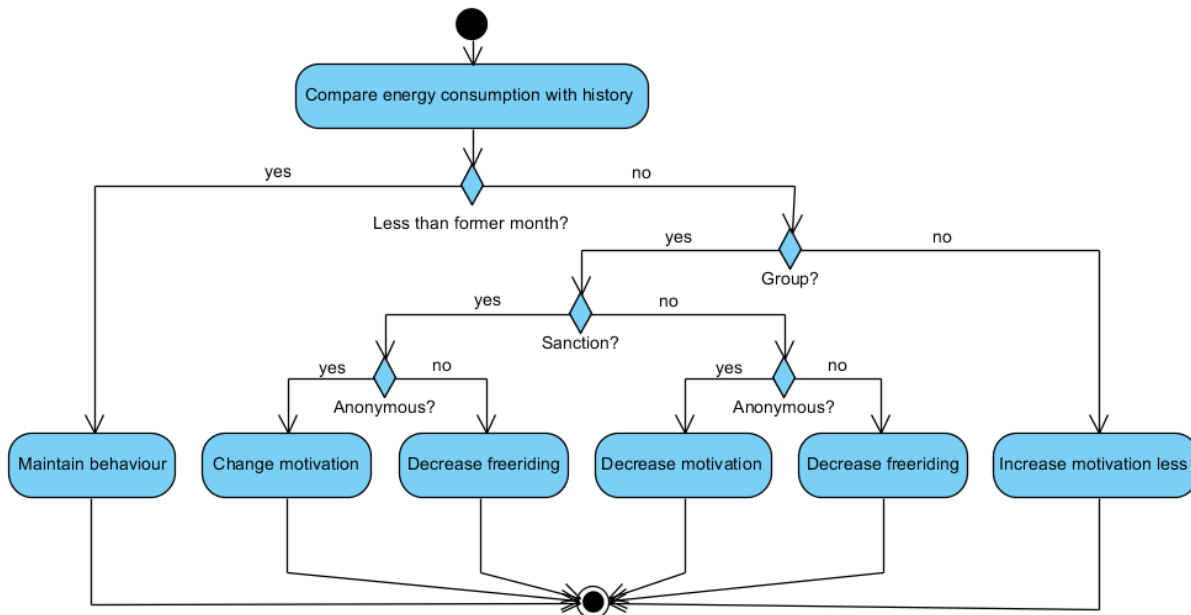


Figure 6: Activity diagram for user agent state "compareWithHistory" [drawn with Visual Paradigm]

### 6.3.8 Defining Interactions

As we saw in section 6.3.5 capturing interactions on a high level can be done using *UML use case diagrams*. Capturing interactions in more detail can be done by using *UML sequence diagrams*. This can be used to further specify use cases that involve direct interactions (usually in form of message passing) between entities (agents and objects).

In Software Engineering *UML sequence diagrams* are used primarily to show the interactions between objects in the sequential order that those interactions occur. Often they depict the actors and objects involved in a specific use case realisation and the sequence of messages exchanged between the actors and objects needed to carry out the functionality of the use case realisation. But sometimes they also capture wider scenarios that go beyond a specific use case. The relevant components of a sequence diagram are listed in Table 8.

Component	Symbol	Description
Lifeline		Named element which represents an individual participant in the interaction
Message		From sender to receiver
Message		Return message
Execution		Represents a period of time in which the participant is active
Message		Self message
Loop		Wrapper for representing loops (has one compartment)
Alternative		Wrapper for representing alternatives (has as many compartments as alternatives exist)

Table 8: Relevant sequence diagram components

In our case we discussed the technical way of implementing the "observe others" use case during one of our *focus group discussions*. Figure 7 shows the sequence diagram we developed during our discussion for this use case. The entities involved are users and units that provide information. Users interact with information units and with each other. Information units interact with the users and with

each other. Creating this diagram sparked a discussion around if we should consider a database that stores historic information in our model or not. It is currently not represented in the scope table (Table 1). In the end we agreed that for our initial model we will leave it out but keep a record of it in the scope table as it might be something we want to consider in the future. We then removed it from the final version of our sequence diagram.

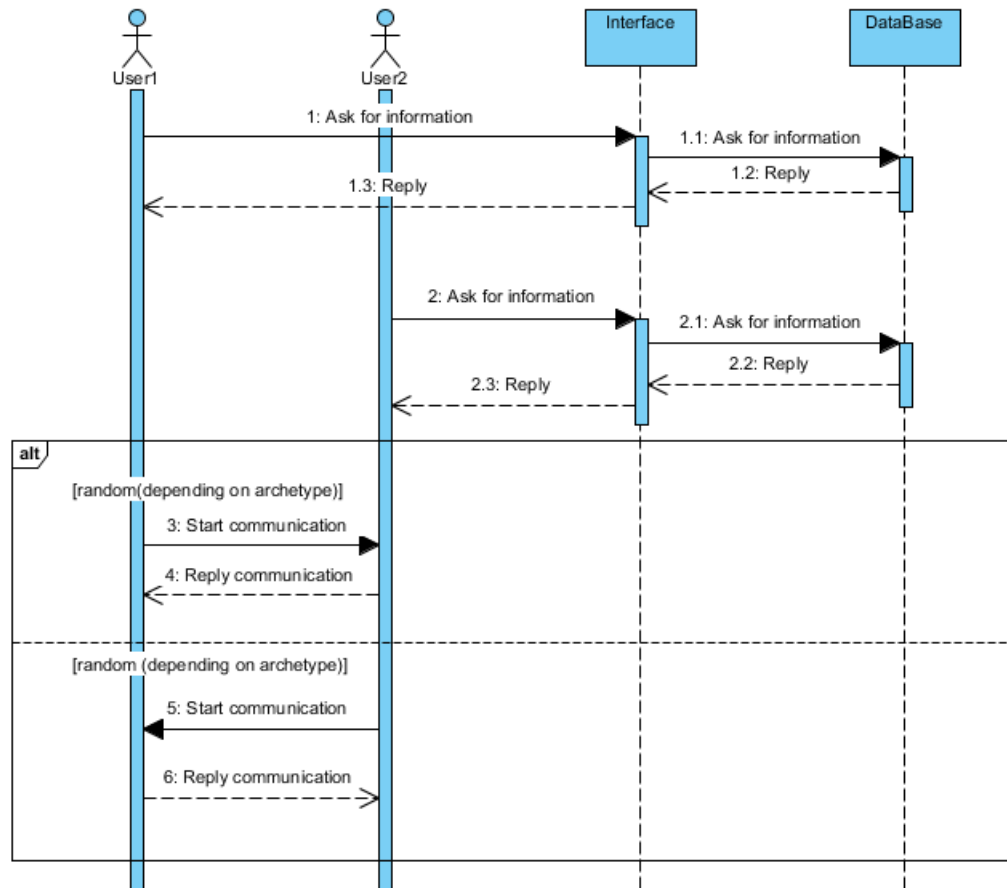


Figure 7: Initial sequence diagram for the use case "observe others" [drawn with Visual Paradigm]

### 6.3.9 Defining the Artificial Lab

Finally we need to define an environment in which we can embed all our entities and define some global functionality. We call this environment our "artificial lab". For the development of our artificial lab we use a class definition as described in Section 6.3.7. Within this class definition we consider things like global variables (e.g. to collect statistics), compound variables (e.g. to store a collection of agents and objects), and global functions (e.g. to read/write to a file). We also need to make sure that we have all variables in place to set the experimental factors and to collect the responses we require for testing our hypotheses. We derive our class content through *focus group discussions*. To inform these discussions we need to look at our list of objectives (see Section 6.3.3) and our scope table (see Section 6.3.4). The final class definition should only contain key variables and functions. Figure 8 shows the "Artificial Lab" class definition for our illustrative example. Variable names including "[]" represent collection variables.

Artificial Lab
-schoolEnergyConsumption
-numEnvironmentalChampions
-numEnergySavers
-numGeneralUsers
-numBigUsers
-isDataApportionmentAvailable
-isApportionmentLevelGroup
-isInformativeFeedbackAvailable
-isAnonymityGiven
-isSanctionImplemented
-users[]
-offices[]
-lights[]
-computers[]
+calculateSchoolConsumption()
+writeDataToFile()
+findOffice()

Figure 8: Artificial Lab class definition [drawn with Visual Paradigm]

Sometimes it can be helpful to create a sequence diagram as described in Section 6.3.8 to visually show the order of execution describing the actions taken on various elements at each step of the simulation from a high level approach. The way and order in which all entities are initialised, as well as the way and order how they are updated and how their interactions are handled, is often not trivial and a major source of artefacts. Therefore, in such a case it needs to be clearly documented and specified. In our illustrative example we do not have any obvious complex order dependencies and therefore it was not necessary to create such a high level sequence diagram.

At this point we have all the information together. Using the collection of diagrams and tables that we produced, the model to be handled should be fully specified and as well understood as it can be without running it. The next step is to take all the information and either start with the implementation or pass on the details to let a professional software developer deal with it.

## 6.4 Conclusion

There seems to be a fear of non-computer scientists when it comes to "formal approaches". This might be due to the fact that formal approaches are often presented in a way that makes modelling a very complex and costly task and that seem to take away opportunities for exploratory model development. While this might be true for very large projects it is usually not the case for smaller ones as tools and techniques do not have to be applied in a dogmatic fashion. They are there to aid the modelling process wherever one thinks it would be appropriate or helpful to use them. Thinking about this as being a *more structured approach* that *adds transparency* to model development rather than a *formal approach* that *makes modelling a complex task* might take away some of the fear. While there will always be a place for informal modelling (in Software Engineering often coined as "fast prototyping" to quickly try out things) we believe that there is also a place for a more structured approach to modelling.

We have found the framework described in the second part of the chapter very helpful in terms of communicating in multi-disciplinary teams during focus group meetings and also for documenting the outcomes of these discussions. It (or parts of it) has been extensively used by the group of PO Siebers (the first author of this chapter) for many different projects ranging from "Studying People Management Practices in Retail" (Siebers and Aickelin 2011) where we worked with colleagues from Economics and Work Psychology and a leading UK retailer to "Simulating Peace Building Activities in Africa" (Siebers et al 2017) where we worked with colleagues from the School of Politics and Psychology. We are currently also applying the framework in several new projects, including industrial partners.



So far the feedback from the participating team members has always been very positive. Using these methods has aided "the fun" of collaborative model development. Using object oriented principles and tools from Software Engineering also helped us to develop simulation models that are easy to maintain and easy to extend. Rather than building a model from scratch every time we start a new study we can reuse previously developed model component with confidence. Using a formal approach to modelling is also a big benefit when it comes to publications as the resulting models are transparent and well documented.

## Further Reading

There is a host of literature on the topic of Software Engineering. A book that provides a comprehensive yet easy to understand entry to most of the Software Engineering topics discussed in this book chapter is Lethbridge and Laganieri (2005). If you are mainly interested in learning more about UML then Fowler (2003) is sufficient. A lot of ideas for ABSS stem from the Computer Science field of Artificial Intelligence, and here in particular Multi-Agent Systems. A good overview on the wide area of topics (including AOSE) is Weiss (2013). Finally, the JASSS special issue "Engineering ABSS" (Siebers and Davidsson 2015) provides lots of information and case studies

## References

- Anderson JR., Bothell D, Byrne MD, Douglass S, Lebiere C, Qin Y (2004). An integrated theory of the mind. *Psychological Review*, 111(4):1036-1060
- Bauer B, Odell J (2005) UML 2.0 and agents: how to build agent-based systems with the new UML standard. *Journal of Engineering Applications of Artificial Intelligence*, 18(2):141-157
- Beck K (2004) *Extreme Programming Explained: Embrace Change*, 2<sup>nd</sup> Edition, Addison Wesley.
- Bedwell B, Leygue C, Goulden M, McAuley D, Colley J, Ferguson E, Spence A (2014). Apportioning energy consumption in the workplace: a review of issues in using metering data to motivate staff to save energy. *Technology Analysis & Strategic Management*, 26(10):1196-1211
- Bergenti F, Gleizes M-P, Zambonelli F (eds.) (2004) *Methodologies and software engineering for agent systems: the agent-oriented software engineering handbook*. Kluwer, Boston
- Bersini H (2012) UML for ABM. *Journal of Artificial Societies and Social Simulation* 15(1)9. <<http://jasss.soc.surrey.ac.uk/15/1/9.html>>
- Boero R, Squazzoni F (2005) Does Empirical Embeddedness Matter? Methodological Issues on Agent-Based Models for Analytical Social Science. *Journal of Artificial Societies and Social Simulation* 8(4)6. <<http://jasss.soc.surrey.ac.uk/8/4/6.html>>.
- Bosse T, Jonker CM, van der Meij L, Treur J (2005) LEADSTO: a language and environment for Analysis of Dynamics by Simulation. Eymann T, Klügl F, Lamersdorf W, Klusch M, Huhns MN (eds.) *Proc. of the 3<sup>rd</sup> German Conference on Multi-Agent System Technologies, MATES'05*. LNCS 3550, Springer, pp. 165-178.
- Bommel P, Müller JP (2007) *An Introduction to UML for Modelling in the Human and Social Sciences*, Phan D, Amblard F (eds.) *Multi-Agent Modelling and Simulation in the Social and Human Sciences*, Bardwell Press, GEMAS Studies in Social Analysis, (Chapter 12)
- Caillou P, Gaudou B, Grignard A, Truong CQ, Taillandier P (2015) A Simple-to-use BDI architecture for Agent-based Modeling and Simulation. *The Eleventh Conference of the European Social Simulation Association (ESSA 2015)*, Sep 2015, Groningen, Netherlands.
- d'Inverno M, Luck M (2001) *Understanding Agent Systems*, Springer-Verlag
- Drogoul A, Vanbergue A, Meurisse T (2003) Multi-agent Based Simulation: Where are the agents? *Multi-agent Based Simulation II, LNCS 2581*, pp 1-15, Springer.

- Drogoul A, Ferber J (1994) Multi-agent simulation as a tool for modelling societies: Application to social differentiation in ant colonies. In Chastelfranchi C, Werner E (eds.) *Artificial Social Systems - 4th Eur. Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'92*, pp. 3-23. Springer.
- Duboz T, Versmisse D, Quesnel G, Muzy A, Ramat E (2006) Specification of dynamic structure discrete event multiagent systems, *Agent-Directed Simulation (ADS 2006)*, Huntsville, AL, USA
- Edmonds B (2004) How formal logic can fail to be useful for modelling or designing MAS. Lindeman G. et al. (eds.) *RASTA 2002, LNAI 2934*, pp 1-15, 2004
- Edmonds B, Moss S (2004) From KISS to KIDS- an 'anti-simplistic' modelling approach. Paul Davidson. et al (eds.) *Multi-Agent Based Simulation, LNAI 3415*. Springer, pp. 130-144.
- Fasli M (2004) Formal Systems  $\wedge$  Agent-Based Social Simulation =  $\perp$ ? *Journal of Artificial Societies and Social Simulation* 7(4)7. <<http://jasss.soc.surrey.ac.uk/7/4/7.html>>
- Fehr E, Fischbacher U, Gächter S (2002) Strong reciprocity, human cooperation, and the enforcement of social norms. *Human nature*, 13(1):1-25.
- Fowler M (2003) *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3<sup>rd</sup> Edition, Pearson Education
- Franchi E (2012) A Domain Specific Language Approach for Agent-Based Social Network Modeling. 2012 IEEE/ACM Int. Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012), Istanbul, Turkey.
- Gamma E, Helm R, Johnson R, Vlissides J (1994) *Design pattern: Elements of reusable object-oriented software*, Addison-Wesley
- Garro A, Parisi F, Russo W (2013) A Process Based on the Model-Driven Architecture to Enable the Definition of Platform-Independent Simulation Models. Pina N, Pacpryzk J, Filipe J (eds.) *Simulation and Modeling Methodologies, Technologies and Applications SIMULTECH 2011 Noordwijkerhout, The Netherlands, July 2011 Revised Selected Papers*, Springer, Berlin, pp. 113-129
- Garro A, Russo W, (2010) easyABMS: A domain-expert oriented methodology for agent-based modeling and simulation. *Simulation Modelling Practice and Theory* 18:1453-1467
- Gilbert N, Troitzsch KG (2005) *Simulation for the Social Scientist*, 2<sup>nd</sup> Edition, Open University Press
- Ghorbani A, Bots P, Dignum V, Dijkema G (2013) MAIA: a framework for developing agent-based social simulations. *Journal of Artificial Societies and Social Simulation*, 16(2):9, 2013
- Ghorbani A, Bots P, Alderwereld H, Dignum V, Dijkema G (2014) Model-driven agent-based simulation: procedural semantics of a MAIA model. *Simulation Modelling Practice and Theory* 49: 27-40.
- Gomez-Sanz JJ, Fernandez CR, Arroyo J (2010) Model driven development and simulations with the INGENIAS agent framework. *Simulation modeling Practice and Theory*, 18(10):1468-1482
- Gomez-Sanz JJ, Fuentes-Fernandez R (2015) Understanding agent-oriented software engineering methodologies. *The Knowledge Engineering Review* 30(4):375-393
- Grimm V, Revilla E, Berger U, Jeltsch F, Mooij WM, Railsback SF, Thulke HH, Weiner J, Wiegand T, DeAngelis DL (2005) Pattern-oriented modeling of agent-based complex systems: lessons from ecology. *Science*, 310(5750): 987-991
- Helleboogh A, Vizzari G, Uhrmacher AM, Michel F (2007) Modeling dynamic environments in multi-agent simulation. *Autonomous Agents and Multi-Agent Systems*, 14(1):87-116
- Himmelspach J, Röhl M, Uhrmacher AM (2010) Component-based models and simulations for supporting valid multi-agent system simulations. *Applied Artificial Intelligence*, 24(5):414-442
- Hocaoglu MF, Firat C, Farjoughian HS (2002) DEVS/RAP: Agent-based simulation. In *Proc. of the 2002 AI, Simulation and Planning in Highly Autonomous Systems conference*, Lisbon, Portugal, IEEE.
- Jones RM (2005). An introduction to cognitive architectures for modeling and simulation. /*Proceedings of the Interservice/Industry Training/Simulation and Education Conference 2005.*/ Orlando, FL.

- Joo J (2013) Perception and BDI Reasoning Based Agent Model for Human Behavior Simulation in Complex System. Kurosu M (ed.) Human-Computer Interaction. Towards Intelligent and Implicit Interaction: 15th Int. Conf., HCI International 2013, Las Vegas, NV, USA, July, 2013, Proc, Part V, Springer Berlin/Heidelberg, pp. 62-71
- Juziuk J, Weyns D, Holvoet T (2014) Design Pattern for Multi-Agent Systems: A Systematic Literature Review. In. Shehory O, Sturm A (eds.) Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages and Frameworks, Chapter 5, pp. 79-99, Springer
- Kasaie P, Kelton WD (2015) Guidelines for design and analysis in agent-based simulation studies. In *Proc. of the 2015 Winter Simulation Conference (WSC '15)*. IEEE Press, Piscataway, NJ, USA, 183-193.
- Kravari K, Bassiliades N (2015) A Survey of Agent Platforms. *Journal of Artificial Societies and Social Simulation* 18 (1) 11 <<http://jasss.soc.surrey.ac.uk/18/1/11.html>>
- Klügl F, Karlsson L (2009) Towards pattern-oriented design of agent-based simulation models, Proc. of the 7th German conference on Multiagent system technologies, September 2009, Hamburg, Germany
- Knublauch H (2002) Extreme Programming of Multi-Agent Systems. In. Proc. of AAMAS 2002, Bologna, pp. 704-711
- Köhler M, Langer R, von Lüde R, Moldt D, Rölke H, Valk R (2007) Socionic Multi-Agent Systems Based on Reflexive Petri Nets and Theories of Social Self-Organisation. *Journal of Artificial Societies and Social Simulation* 10(1)3 <<http://jasss.soc.surrey.ac.uk/10/1/3.html>>.
- Kubera Y, Mathieu P, Picault S (2011) IODA: an interaction-oriented approach for multiagent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23(3):303-343
- Laird JE, Newell A, Rosenbloom PS (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1-64.
- Law AM (2007). *Simulation Modeling & Analysis*, McGraw-Hill, 4<sup>th</sup> edition.
- Lethbridge TC, Laganieri R (2005) *Object-Oriented Software Engineering: Practical Software Development Using UML and Java: Practical Software Development*, McGraw Hill
- Mascardi V, Martelli M, Sterling L (2004) Logic-based specification languages for intelligent software agents. *Theory Pract. Log. Program.* 4(4):429-494.
- McGarty G, Yzerbyt VY, Spears R (2002). Social, cultural and cognitive factors in stereotype formation. McGarty G, Yzerbyt VY, Spears R (eds.) *Stereotypes as explanations*. New York: Port Chester, 1-15.
- Mitleton-Kelly E (2003). Complexity Research - Approaches and Methods: The LSE Complexity Group Integrated Methodology. A Keskinen, M Aaltonen, E Mitleton-Kelly (eds.) *Organisational Complexity*. Tutu Publications. Finland Futures Research Centre, Turku School of Economics and Business Administration, Turku, Finland, pp. 56-77.
- Moyo D, Ally AK, Brennan A, Norman P, Purshouse RC, Strong M (2015) Agile Development of an Attitude-Behaviour Driven Simulation of Alcohol Consumption Dynamics. *Journal of Artificial Societies and Social Simulation* 18 (3) 10 <<http://jasss.soc.surrey.ac.uk/18/3/10.html>>
- Nikolai C, Madey G. (2008) Tools of the Trade: A Survey of Various Agent Based Modeling Platforms. *Journal of Artificial Societies and Social Simulation*, 12(2)2 <<http://jasss.soc.surrey.ac.uk/12/2/2.html>>
- Norling E (2003). Capturing the quake player: Using a BDI agent to model human behaviour. In Rosenschein JS, Sandholm T, Wooldridge M, Yokoo M (eds.) Proc. of the 2<sup>nd</sup> Int. Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), Melbourne, pages 1080-1081.
- North MJ, Macal CM (2011) Product design patterns for agent-based modeling. Jain S, Creasey R, Himmelspach J, White KP, Fu MC (eds.) Proc. of the Winter Simulation Conference (WSC '11), pp 3087-3098.
- Odell J, Parunak HVD, Bauer B (2000) Extending UML for Agents. Lesperance Y, Yu E (eds.) Proc. of the Agent-Oriented Information Systems Workshop at the 17th NCAI, pp. 3-17, 2000.
- Ostrom E (2005) *Understanding institutional diversity*. Princeton University Press.

- Ozik J, Collier N, Combs T, Macal CM, North M (2015) Repast Symphony Statecharts. *Journal of Artificial Societies and Social Simulation* 18 (3) 11 <<http://jasss.soc.surrey.ac.uk/18/3/11.html>>
- Pyritz B (2003) Craftsmanship versus engineering: Computer programming—An art or a science?. *Bell Labs Tech. J.*, 8: 101-104
- Railsback SF, Lytinen SL (2006) Agent-based simulation platforms: review and development recommendations. *Simulation*, 82: 609-623
- Richiardi M, Leombruni R, Saam NJ, Sonnessa M (2006). A Common Protocol for Agent-Based Social Simulation. *Journal of Artificial Societies and Social Simulation* 9(1)15 <<http://jasss.soc.surrey.ac.uk/9/1/15.html>>
- Robinson S (2004). *Simulation: The practice of model development and use*. John Wiley & Sons: Chichester, UK
- Rossiter S (2015) Simulation Design: Trans-Paradigm Best-Practice from Software Engineering. *Journal of Artificial Societies and Social Simulation* 18 (3) 9 <<http://jasss.soc.surrey.ac.uk/18/3/9.html>>.
- Scherer S, Wimmer M, Lotzmann U, Moss S, Pinotti D (2015) Evidence Based and Conceptual Model Driven Approach for Agent-Based Policy Modelling, *Journal of Artificial Societies and Social Simulation* 18(3)14 <<http://jasss.soc.surrey.ac.uk/18/3/14.html>>.
- Shannon RE (1998) Introduction to the Art and Science of Simulation. Medeiros DJ, Watson EF, Carson JS, Mannivannan MS (eds.) *Proc. Of the 1998 Winter simulation Conference*, pp. 7-14
- Siebers PO, Davidsson P (2015) Engineering Agent-Based Social Simulations: An Introduction (Special Issue Editorial). *Journal of Artificial Societies and Social Simulation*, 18(3) <<http://jasss.soc.surrey.ac.uk/18/3/13.html>>.
- Siebers PO, Aickelin U (2011). A First Approach on Modelling Staff Proactiveness in Retail Simulation Models. *Journal of Artificial Societies and Social Simulation* 14(2)2 <<http://jasss.soc.surrey.ac.uk/14/2/2.html>>
- Siebers PO, Onggo BSS (2014) Graphical Representation of Agent-Based Models in Operational Research and Management Science using UML. In. *Proc. Of the Operational Research Society Simulation Workshop 2014 (SW14)*, pp. 143-155
- Siebers PO, Figueredo GP, Hirono M, Skatova A (2017) Developing Agent-Based Simulation Models for Social Systems Engineering Studies: A Novel Framework and its Application to Modelling Peacebuilding Activities. Garcia-Diaz C, Olaya Nieto C (eds.) *Social Systems Engineering: The Design of Complexity*. John Wiley & Sons.
- Sommerville I (2016) *Software Engineering*, 10<sup>th</sup> Edition, Pearson
- Stahl T, Voelter M, Czarnecki K (2006) *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.
- Susanty M (2015) Adding psychological factors to the model of electricity consumption in office buildings. MSc Dissertation, Nottingham University, School of Computer Science.
- Taatgen NA, Lebiere C, Anderson JR (2006). Modeling Paradigms in ACT-R. Sun R (ed.) *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*. Cambridge University Press; pp 29-52.
- Weiss G (2013) (ed.) *Multiagent Systems*, 2<sup>nd</sup> Edition, MIT Press, Cambridge
- Weyns D, Holvoet T (2004) A formal model for situated multi-agent systems. *Fundamenta Informaticae*, 63(2-3):125-158
- Winikoff M, Padgham L (2013) Agent-Oriented Software Engineering. Weiss G (ed.) *Multiagent Systems*, 2<sup>nd</sup> Edition, Chapter 15, pp. 695-758, MIT Press, Cambridge
- Wooldridge M (2009) *An Introduction to MultiAgent Systems*, John Wiley & Sons
- Wray RE, Laird JE, Nuxoll A., Stokes D., Kerfoot A. (2005) Synthetic adversaries for urban combat training. *AI Magazine*, 26(3):82-92

- Wray RE, Jones RM (2005) An introduction to Soar as an agent architecture. Sun R (ed.) *Cognition and Multi-agent Interaction: From Cognitive Modeling to Social Simulation*, Cambridge University Press, pp 53-78
- Zeigler BP (1990) *Object Oriented Simulation with Hierarchical Modular Models: Intelligent Agents and Endomorphic Systems*. Academic Press
- Zhang T, Siebers PO, Aickelin U (2011) Modelling electricity consumption in office buildings: An agent based approach. *Energy and Buildings*, 43(10), 2882-2892.