

Design Patterns for Multilevel Modeling and Simulation

Luca Serena*, Moreno Marzolla*†, Gabriele D’Angelo*†, Stefano Ferretti‡

*Department of Computer Science and Engineering, University of Bologna, Italy

†Center for Inter-Department Industrial Research ICT, University of Bologna, Italy

‡Department of Pure and Applied Sciences, University of Urbino Carlo Bo, Italy

{luca.serena2,moreno.marzolla,g.dangelo}@unibo.it, stefano.ferretti@uniurb.it

Abstract—Multilevel modeling and simulation (M&S) is becoming increasingly relevant due to the benefits that this methodology offers. Multilevel models allow users to describe a system at multiple levels of detail. From one side, this can make better use of computational resources, since the more detailed and time-consuming models can be executed only when/where required. From the other side, multilevel models can be assembled from existing components, cutting down development and verification/validation time. A downside of multilevel M&S is that the development process becomes more complex due to some recurrent issues caused by the very nature of multilevel models: how to make sub-models interoperate, how to orchestrate execution, how state variables are to be updated when changing scale, and so on. In this paper, we address some of these issues by presenting a set of design patterns that provide a systematic approach for designing and implementing multilevel models. The proposed design patterns cover multiple aspects, including how to represent different levels of detail, how to combine incompatible models, how to exchange data across models, and so on. Some of the patterns are derived from the general software engineering literature, while others are specific to the multilevel M&S application area.

Index Terms—Multilevel modeling, Design patterns, Agent-based models, Multiscale simulation.

I. INTRODUCTION

Modeling and simulation are powerful tools that are used to study the behavior of a complex system without the need of conducting experiments on the “real thing”. Over the years, many modeling methodologies have been developed and are routinely used; different research communities tend to have their own preferred set of tools, e.g., Agent-Based Models (ABMs) are frequently used in the social sciences, whereas continuous (equation-based) models are frequently used to study the diffusion of epidemics [1].

The most frequently used class of models are monolithic, meaning that a single model takes care of the whole system

This is the author’s version of the article: “Luca Serena, Moreno Marzolla, Gabriele D’Angelo, Stefano Ferretti, Design Patterns for Multilevel Modeling and Simulation, proc. 2023 IEEE/ACM 27th International Symposium on Distributed Simulation and Real-Time Applications (DS-RT’23), Singapore, October 4–5, 2023, pp 48–55”. ©2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The publisher version of this paper is available at <https://dx.doi.org/10.1109/DS-RT58998.2023.00015>.

(models that have some internal structuring that merely derives from software engineering good practices of decomposition are still considered “monolithic”). Although monolithic models are appropriate for many kinds of studies, they fall short when large, complex scenarios must be investigated. Firstly, the possibility of combining existing models can reduce development time and simplify verification and validation, provided that the sub-models have already been properly validated. Secondly, complex scenarios may be too large and/or too complex to be evaluated at the maximum level of detail. These facts have motivated the use of *multilevel models*, sometimes referred to as *multilayer* or *multi-resolution* models.

Multilevel models employ multiple sub-models that may or may not be active at the same time; sub-models can be based on different paradigms (continuous, discrete, agent-based, stochastic, and so forth), and may describe the system (or part thereof) at different levels of detail. One key aspect of multilevel modeling is that the decomposition into sub-models does not need to be static, as Figure 1 depicts. In particular, multilevel models are allowed to: (i) switch any part of the system under study to a different type of model (e.g., from continuous to discrete models or back; Figure 1.a), (ii) dynamically change the spatial resolution of any part of the model (Figure 1.b), (iii) dynamically change the time resolution of any part of the model (Figure 1.c), or (iv) dynamically change the amount of state variables, i.e., the accuracy (Figure 1.d).

Multilevel techniques allow multiple level of details to be used either for different portions of the system under study, or at different points in simulated time. The goal is to make better use of available computational resources: indeed, large and complex models might not be executed efficiently if they are represented at the maximum possible level of detail. Even when they could, they would produce a large amount of data that, for the most part, would probably be unimportant. Taking as an example a multilevel traffic model, we might employ an equation-based representation of the aggregate flow of vehicles, and switch to a more accurate agent-based model on areas where interesting patterns emerge, such as road congestion; this allows the simulation to “zoom in” and study in detail how congestion develop and resolve. As soon as the traffic is back to normal, the model can switch back to the faster, but less accurate, equation-based model.

In the last decades many applications employing multilevel

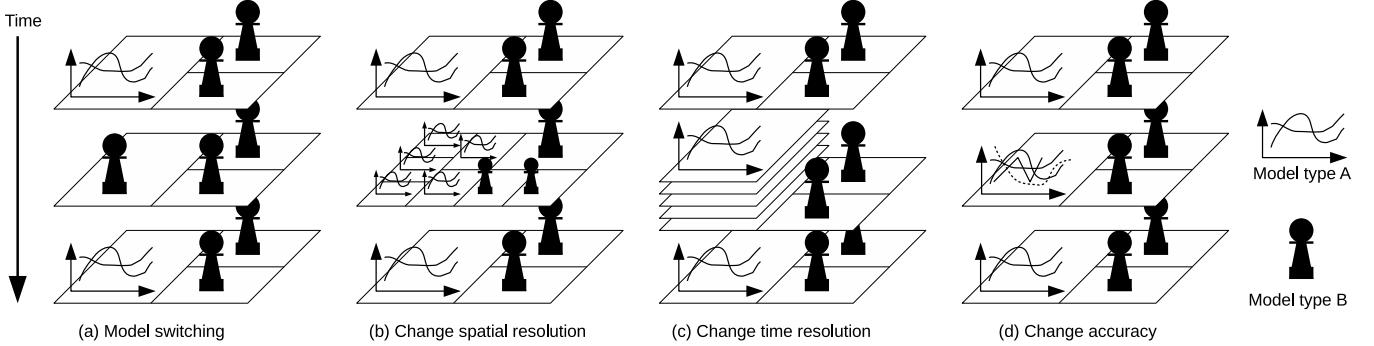


Fig. 1: The four main realizations of multi-level modeling.

frameworks have been used in several fields of study, e.g., human mobility, traffic modeling, urban planning, social sciences, and others (see Section II). Despite the advantages outlined above, multilevel modeling brings some issues that must be addressed: how can sub-models interact? How can sub-models be scheduled for execution to make efficient use of the available computational resources? How can sub-models be orchestrated?

The analysis of the scientific literature reveals that there are some recurrent solutions to the issues above. In this paper, we summarize these solutions into a set of design patterns, with the aim of contributing towards a wider adoption of multilevel M&S. Some of the design patterns come directly from the software engineering domain; others are specific to the multilevel M&S domain. Although reasonably complete, due to space constraints the collection of patterns described here is not meant to be exhaustive; however, we believe that it covers the most important aspects.

This paper is structured as follows: in Section II we introduce the concepts of multilevel modeling and design pattern. Then, from Section III to Section VII we describe the design patterns grouped by function. Finally, some concluding remarks are discussed in Section VIII.

II. BACKGROUND

Several modeling and simulation paradigms exist. They can be classified according to at least two dimensions: (i) how the state space is represented, and (ii) how time is represented.

In *continuous-space models* the state space is represented by continuous values (e.g., real numbers). Conversely, in *discrete-space models* the state space is described using discrete values; a classical example is discrete Cellular Automata (CA), where each cell can assume a finite set of values. In some cases, both continuous and discrete variables are used to represent the state space; in this case we have *mixed models*.

For what concerns time management, we similarly have *continuous-time* and *discrete-time* models [2]. In continuous-time models, the state space changes continuously over time; these models usually are based on sets of Ordinary Differential Equations (ODEs). In discrete-time models, the state space is updated only at specific points in time. Additionally, there are *time-stepped* models, where the time is divided into regular

discrete time frames, each one potentially corresponding to a defined length of time. Finally, *stochastic models* such as Monte Carlo simulations do not rely on any representation of time, since they are concerned with the computation of numerical results using stochastic processes involving a large number of simulated experiments.

Different choices of space and time representation lead to different types of models; to cite a few:

- Equation-Based Models (EBMs) are usually continuous-space, continuous-time and based on differential equations [3]; EBMs can usually be handled efficiently and are well suited for describing aggregate parameters over very large populations of entities.
- ABMs are usually continuous-space, discrete-time and consist of a collection of autonomous, interacting computational objects (agents) that are situated in space and time [4]. ABMs are well suited for representing systems where interactions among agents, and between agents and the environment, are particularly important. However, ABMs can be computationally demanding, particularly if the number of agents is high.
- CA are usually discrete-space, time-stepped (although all possible variants exist); CA represent the domain as a lattice of cells with simple rules to update the state of each cell based on the state of a subset of neighbors. Many classes of CA can be evaluated quite efficiently through parallel computations.

In multilevel models, different paradigms can coexist [5]; sub-models might be either semantically distinct (e.g., an urban simulation where different components describe pedestrian mobility, traffic flows, air pollution, land use and so forth), or describe the same item at different levels of detail (e.g., traffic models that switch from EBMs for aggregate traffic flows and detailed ABMs that describe individual vehicles) [6].

The application spectrum of multilevel M&S techniques is very wide. In biology, chemistry or material science macroscale continuum models can be used to simulate the behavior of fluids, solids, and other materials, molecular simulators can study the dynamics and the interactions of atoms and molecules, while an intermediate scale can be used to simulate the behavior of larger groups of molecules, such

as polymers or proteins [7], [8]. In crowd and traffic simulation, usually an ABM describes the behavior of individual pedestrians and vehicles [9], while a macroscopic model deals with equations that describe an aggregate high-level view of the system [10], [11]. Finally, a recurrent scheme to study the diffusion of epidemics is to have within-host models that describe pathogen-host interactions, taking into consideration immune system responses and the effect of therapies, and between-host models that capture the dynamics of the infection as it spreads from individual to individual [12]–[14].

Multilevel techniques simplify the development of complex models, because they allow code reuse from existing sub-models, cutting down development and validation times. Another key aspect is the possibility of changing the level of detail or the type of paradigm at run-time, to get a suitable trade-off between computational efficiency of coarse-grained models and accuracy of fine-grained representations. Indeed, it is often the case that only some critical parts of a simulation are worth being represented at a high level of accuracy, so using the maximum level of detail everywhere might be a waste of time and resources.

However, multilevel M&S techniques raise several issues, some of which are concerned with software engineering aspects (e.g., how to integrate components that were not necessarily created to work together), while others are domain-specific (e.g., how to ensure consistency among sub-models). Although the concrete solutions of these issues are problem-specific, there are some recurrent patterns that are frequently used in the literature.

In software engineering, *design patterns* are standardized and reusable solutions to recurrent software design problems that have been proven to work effectively in practice. Design patterns are not algorithmic solutions; rather, they are abstract descriptions of solution schemes to classes of problems that must be instantiated to each specific problem.

Conceptualized for the first time by Gamma et al. [15], software design patterns have become an important tool for helping developers to design high-quality, maintainable, and efficient software systems.

In this paper, we describe some design patterns that are found in multilevel M&S applications. The patterns are classified into five categories, as illustrated in Figure 2:

- *Orchestration patterns* deal with the flow of execution of sub-models (Section III).
- *Structural patterns* describe how sub-components can be aggregated into complex models (Section IV). Note that these patterns are taken directly from [15], since they are relevant for multilevel modeling besides general Object-Oriented programming.
- *Execution policy patterns* specify the mapping between components and execution units (Section V).
- *Information exchange patterns* define how data can be transferred between models of different types, e.g., continuous and discrete-space models (Section VI).
- *Multiscale patterns* define how models employing different levels of details can be integrated (Section VII).

III. ORCHESTRATION PATTERNS

Multilevel models involve the execution of multiple components, that may be of different types (e.g., continuous and discrete models), or of the same type using different parameters (e.g., different time-steps). The components may be organized arbitrarily, i.e., not necessarily in a strict hierarchy.

Orchestration patterns define how execution is passed from one component to another. In the *Model's Controller* pattern, sub-models are executed by an external entity called *Controller*, who acts as an interface to the user. In the *Director-Worker* pattern, control is passed from the active component to a different one. Finally, the *Concurrent Modularity* pattern does not assume a strict hierarchical structuring of sub-modules, and allows components to interact in a peer-to-peer way.

The *Director on Hold* and *Worker on Demand* patterns are possible realization of the Director-Worker paradigm. In the Director on Hold realization, the Director is suspended until the worker(s) terminates execution. The Worker on Demand pattern pre-allocates the pool of workers in order to avoid the overhead of dynamically creating/destroying them.

A. Models' Controller

In this pattern there exists one entity, the Controller, that is in charge of (i) acting as the interface to the user or a higher-level model; (ii) scheduling the execution of the various sub-models; (iii) keeping a global state; (iv) managing the exchange of information among sub-models (see top of Figure 3).

The presence of a Controller has the advantage of centralizing the scheduling and management logic, therefore allowing separation of concerns between functionality and implementation. It also allows more flexibility, as adding an additional sub-model is somewhat easier, since only the Controller is involved. An obvious disadvantage is that the Controller might become very complex if a large number of incompatible sub-models are used.

The Controller pattern has been used in [16] to investigate crowd evacuation using a multilevel model. The model relies on a synchronization module to schedule execution of micro and macro scales and manage the exchange of information.

B. Director-Worker

The Director-Worker pattern (bottom part of Figure 3) relies on a hierarchical structuring of sub-models. Each sub-model can act as a worker with respect to its parent module, and as a director with respect to children modules (if any). Control is passed from a Director to a Worker. The Director implements some of the functionalities of the Controller above; however, unlike the Controller, a Director is itself a sub-model, whereas the Controller is an external entity that is not part of the model. The Director-Worker pattern can be combined with the Composite pattern (see Section IV).

The Controller and Director-Worker patterns are not mutually exclusive. For instance, in [17] there is a simulator at the top of the hierarchy that relies on a wrapper script to manage

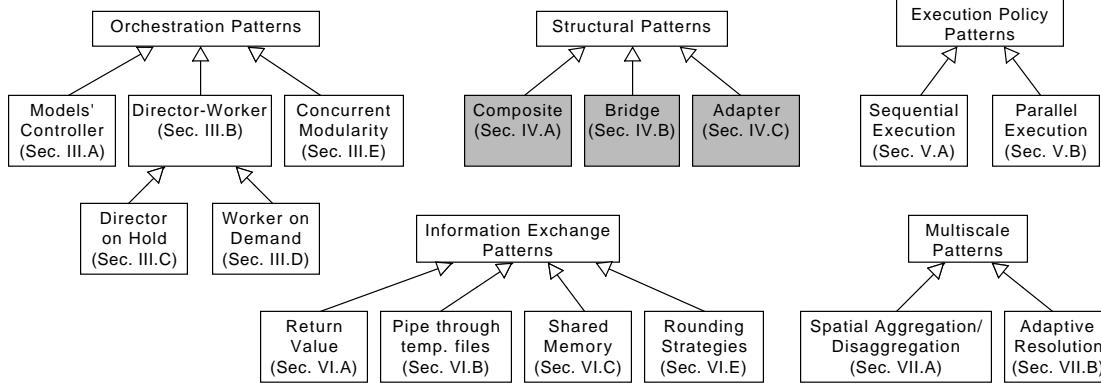


Fig. 2: Summary of the patterns described in this paper. Those originally described in [15] are shaded.

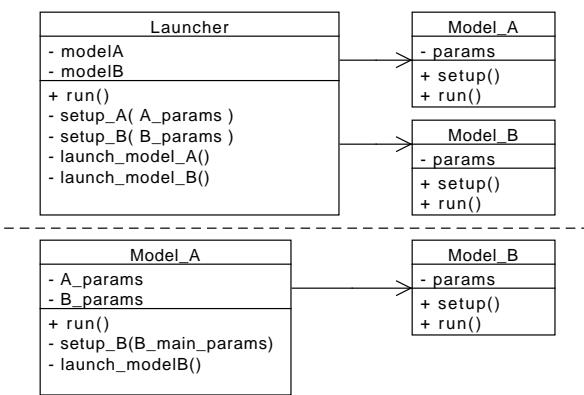


Fig. 3: Class diagrams of the Model's Controller (top) and Director-Worker pattern (bottom).

the various instances of the underlying models. Thus, in this case the wrapper script can be considered both as a Worker in a Director-Worker scheme and as a Models' Controller of the lower-level modules.

C. Director on Hold

The *Director on Hold* pattern is the simplest realization of the Director-Worker paradigm: the Director instantiates new Workers when needed, and suspends itself while the Workers are active. At the end, Workers are terminated and the Director resumes execution. Although very simple, this strategy may incur a significant overhead if creation/destruction of Workers is a costly procedure. Indeed, while EBMs might be cheaper to build and destroy, the same cannot be said for ABMs, as the creation of the agents and the storage of their state is often a non-negligible activity. Furthermore, the Director does not execute any computation while the Workers are active, therefore reducing the level of concurrency that might be allowed by the model.

D. Worker on Demand

This pattern addresses one of the limitations of the Director on Hold pattern, namely, the overhead of creating/destroying Workers when needed. In the Worker on Demand pattern, as shown in Figure 4, all workers are created at the beginning of the execution and are kept in stand-by; when one or more Workers are required, those in the pool are dynamically assigned to complete some task.

The Worker on Demand pattern separates initialization of Workers from the execution of tasks, with two main benefits: complex entities are created only once (in case there is a large number of entities this may save significant time during the life of the model), and the state of the entities can be stored and retrieved for additional examination. The drawback, however, is that the pool of Workers takes up memory space even when inactive, making this strategy not applicable in memory-constrained environments.

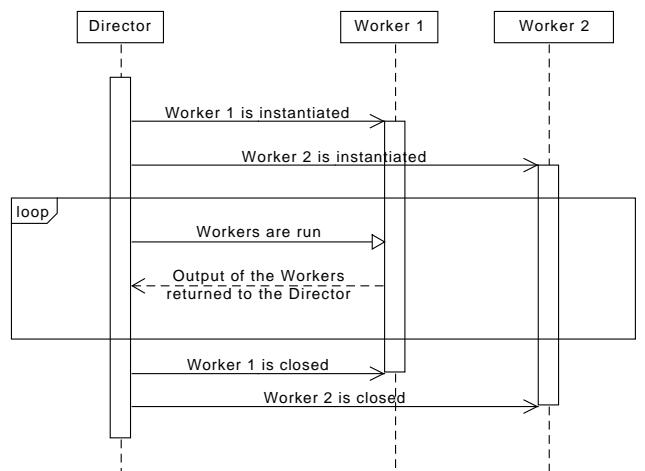


Fig. 4: Sequence diagram of Worker on Demand.

E. Concurrent Modularity

As already discussed, it is convenient to run multiple model instances in parallel when possible. The *Concurrent Modularity* pattern allows concurrent execution of semantically different models, possibly of different types. For example, a multilevel urban traffic model might include sub-models for vehicle movements, pedestrian movements, air pollution, and sound pollution. These sub-models might be executed concurrently, provided that interactions are properly accounted for (see Section VI).

The Concurrent Modularity pattern entails a thorough time management, since different (sub-)models may use different time granularity and/or different concepts of time; the latter happens, for example, when one mixes continuous and time-stepped models.

The issue of time management can be addressed in different ways, such as:

- A time translation mechanism, where the local time of a sub-model is translated into a global time understood by all other components.
- Checkpointing, where sub-models proceed in lockstep and are synchronized periodically. A sub-model that reaches a checkpoint stops execution, and resumes when all other sub-models have also reached the same checkpoint.
- Rollback mechanisms, where inconsistencies in state updates are detected and undone by rolling back the (virtual) simulation time to a previous time where a consistent state were computed [18].

IV. STRUCTURAL PATTERNS

Structural patterns describe how software elements can be composed into larger structures while promoting flexibility and code maintainability. The patterns described in this section are taken from [15], where they have been initially proposed in the context of software engineering.

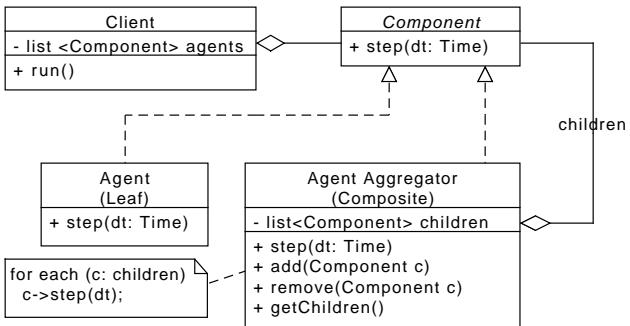


Fig. 5: UML class diagram of Composite pattern applied to a multiscale ABM scenario.

A. Composite

The Composite pattern enables the hierarchical composition of objects according to a tree-like structure, allowing atomic

and composite objects to be treated uniformly. This makes it easy to add new types of objects to a system, as the interface for all components remains the same. The pattern is composed of three main elements (see Figure 5):

- *Component*, an interface that defines the common methods for both the Leaf and the Composite.
- *Leaf*, an end node of a tree structure.
- *Composite*, an internal node of a tree structure.

An application of this design pattern is in the context of hierarchical ABMs. Here, composite objects are actors at the macro level that act as container of agents at the micro level. The composite object may therefore represent a portion of a model that is represented at a coarser level of detail; when more accuracy is required, the composite executes the low-level agents that it contains, which in turn might be composite objects and contain some finer-grain sub-models.

This pattern could be applied in [19], where the authors studied the spread of black rats by means of commercial transportation. In this work, the main building block of the simulator is represented by the concept of *World*, defined as a complete and self-sufficient sub-model with its own places, agents, spatial resolution and temporal scale. Worlds can possibly be nested, representing part of another World at greater level of detail. The Composite pattern could be then applied in order to provide a high-level management of all the worlds in the system.

B. Bridge

The Bridge pattern allows developers to separate the abstraction (interface) from the implementation [15]. The Bridge pattern is composed of three main components:

- *Abstraction*, which defines the high-level interface that clients will use.
- *Implementor*, which serves as an interface for describing the technical functionalities of the Abstraction.
- *Concrete Implementor*, which defines the concrete implementation of the Implementor.

Separating the implementation from the interface is one of the cornerstones of Object-Oriented programming; among other things, it allows different implementations of some abstract object to be interchanged, even at run-time, without the need to modify clients that are using the abstract objects. The Bridge pattern can be applied in ABMs, enabling to separate the definition of the agents from the code that defines the behavior of certain types of simulated entity, as shown in Figure 6. For example, in a simulation we could have different types of human agents (the Abstraction), characterized by different types of behavior in response to certain events. Bridge can find an application also in the context of multiscale modeling, for instance when individual and aggregate agents coexist in the same simulated environment like in [20], where the movement of pedestrians is simulated dealing with a hierarchy of crowds, groups, and individuals. This pattern allows the developers to make changes to the behavior of the agents without affecting how the Client interacts with them.

Also, by abstracting away the details of the implementation, the Abstraction provides a simpler interface for the Client, improving the management of the agents at a high-level.

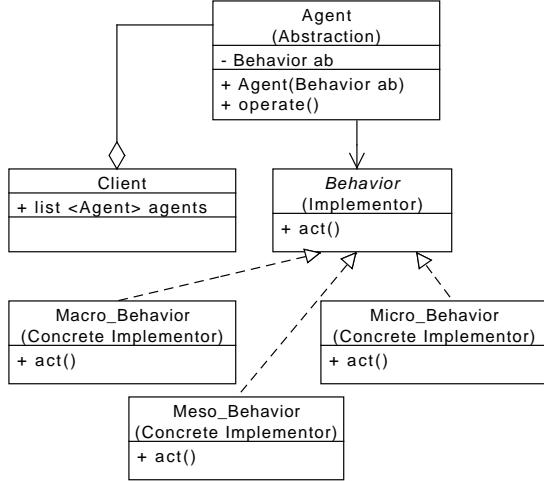


Fig. 6: Class diagram of the Bridge pattern applied to a multiscale ABM scenario.

C. Adapter

The *Adapter* pattern allows two or more components with incompatible interfaces to work together by creating an adapter that converts one interface to another [15]. The Adapter pattern is composed of four components:

- *Client*, the object that uses the Adapter to interact with the Adaptee.
- *Adaptee*, the object that needs to be adapted to work with the Client.
- *Target*, the object that the Client wants to use.
- *Adapter*, the object that acts as an intermediary between the Client and the Adaptee. The Adapter translates the interface of the Adaptee to the interface expected by the Client (i.e., the Target interface).

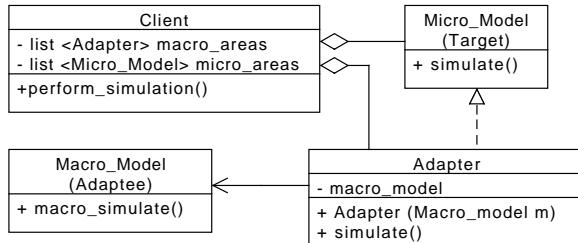


Fig. 7: UML class diagram of the Adapter pattern applied to a multiscale modeling scenario.

Since multilevel M&S is based upon multiple cooperating components (sub-models), the Adapter pattern is extremely useful when the sub-models were not intended to work together. This is usually the case when the multilevel model

is built upon existing components. Another use case for the Adapter pattern is shown in Figure 7, and involves a multiscale traffic model. In these scenarios, the space is often split into micro zones where the behavior of vehicles is modelled individually, and macro zones where the traffic is represented in terms of aggregate metrics such as density, average speed and traffic flow. Suppose that a traffic model was initially developed using one level only, e.g., using only micro zones. Then, the Adapter pattern can be employed to make the macro zones (the Adaptees) look like micro zones from the point of view of the coordinator (the Client).

V. EXECUTION POLICY PATTERNS

Execution policies refers to rules and guidelines that govern the mapping of the model to the underlying execution unit(s). The choice of execution policy depends on several factors, such as the size and complexity of the model, the amount of computational resources available, and the desired level of performance that should be achieved.

A. Sequential Execution

In a sequential execution pattern, the model is executed on a single execution unit, e.g., a single processor or processor core. This is the most common approach, either because the vast majority of models are inherently sequential, or because the existing implementations are. Sequential execution is appropriate when the time required to analyze a model is not the bottleneck.

B. Parallel Execution

Parallel execution involves splitting the model into smaller parts that can be executed simultaneously on multiple execution units, possibly over multiple interconnected machines. Parallel execution has traditionally been applied to speed up the execution of monolithic models, e.g., by processing events in parallel during discrete-event simulations [21] or by employing parallel solvers for analyzing large sets of differential equations.

In the context of multilevel M&S, parallelization may be useful when some form of domain partitioning is used to split the simulation space into separate partitions, each one being modeled at a different level of granularity and/or using different types of models. If the partitions are independent, they can be evaluated in parallel, although in practice the level of parallelism might be reduced because neighboring partitions could need to exchange data periodically, or some form of global consistency must be ensured.

VI. INFORMATION EXCHANGE PATTERNS

Information Exchange patterns define how data can be exchanged between sub-models. Different factors can affect the way in which the information is transferred, such as the type of information (i.e., discrete vs continuous), or the relationship between the involved components (i.e., hierarchical vs peer-to-peer).

A. Return Value

The *Return Value* pattern is the most trivial way to exchange data in a Director-Worker scenario. The Director calls The Worker that returns back to the Director some result. This strategy is very simple but is difficult to implement if the Director is allowed to execute concurrently with the Workers. In this scenario, the execution of Workers is an asynchronous operation that may return well before the Workers terminate. This problem can be addressed using the *futures* pattern used in concurrent programming [22], where the Worker returns a object representing the “promise” to compute a result; the Director will block if it tries to read the result when it has not been computed yet.

B. Pipe through Temporary Files

Another way to exchange information is through temporary files. This provides the following advantages:

- *Flexibility*: it is easy to add fields within the list of values to be returned.
- *Versatility*: it can be used also for flat models, since the consumers only need to know the location of the data file(s), and possibly when new data are available.
- *Data organization*: developers may choose the most appropriate data representation (e.g., JSON, XML, ...) depending on the type of information that needs to be stored and the requirements of the producers and consumers.

However, also known limitations must be considered:

- File operations (creation, read, write) entails some overhead, which can be relevant to a greater or lesser extent depending on the number of operations.
- If the application aborts, the user must ensure that all temporary files are deleted before the simulator is launched again, to avoid pollution of a new execution with stale information.
- Some additional mechanism must be put in place to inform the consumers when new data is available.

C. Shared Memory

In the *Shared Memory* pattern, data is stored in some memory region that is make accessible by all sub-models. Although this is somewhat more efficient than using temporary files, shared memory needs to be emulated on distributed-memory architectures. Furthermore, if shared memory is used for two-way communication, special care should be taken to avoid read-write conflicts.

D. Rounding Strategies

Exchanging data between sub-models that are base on continuous and discrete state representations is a common problem in multilevel modeling. In such scenarios, rounding strategies must be carefully defined, in order to ensure global consistency properties such as conservation of some model-specific entities.

As a practical example, let us consider a multilevel epidemic model where a set of ODEs is used to predict the diffusion

of a contagious disease through a population. To better study the contagion in critical zones such as schools, hospitals, or crowded areas, the model might delegate some regions to more detailed ABMs. In this scenario, it is essential that the total population is conserved, i.e., assuming that the system is closed, the sum of susceptible, infected, recovered and removed individuals must remain the same as time progresses. However, ODE-based models are continuous while ABMs are discrete, so we need to ensure that the values of state variables – in this case, the number of individuals in each of the four classes above – are stationary.

Several rounding strategies have been proposed in the literature: for example, the model could keep track of rounding errors and add/remove individuals in the ABMs when needed.

VII. MULTISCALE PATTERNS

Multiscale patterns deal with the issues related to representing the same sub-model at different levels of detail. The use of multiscale methodologies is motivated by the need to choose an optimal trade-off between execution time and precision of the results.

A. Spatial Aggregation-disaggregation

A recurrent scheme of multiscale model is to have a microscopic level where the behavior of the involved entities is described at a high level of details, and a macroscopic level that deals with aggregate metrics. The two levels might employ different types of models (e.g., ABM for the micro level, and equation-based model for the macro level), or the same type of model with different parameters (e.g., an equation-based model for both levels using a finer time/space subdivision to increase accuracy). The micro level is used when/where “interesting” phenomena emerge; for example, in a large urban traffic model, the micro level would be used to focus on traffic jams or transient congestion zones, in order to study how these pattern form.

Aggregation and disaggregation is a modeling pattern that involves collapsing a large number of entities at the micro level to build a single entity at the macro level (aggregation), and the opposite act of creating multiple entities at the micro level to represent a single entity of the macro level (disaggregation). Therefore, the Aggregation/Disaggregation design pattern establishes the rules by which it is possible to switch between two levels of detail. This pattern is often used in the context of ABM, although in principle it can be applied to other types of models as well.

Multiple realizations of this design patterns have been described in the scientific literature [23]:

- *Zoom* pattern, where the micro entities are destroyed when transitioning into the macro zones, and their information is lost.
- *Puppeteer* pattern, where the micro entities are not destroyed but frozen and temporarily controlled by the macro agents. Micro entities are still able to update their internal state according to their own dynamics, but cannot

autonomously perform actions, which are delegated to the macro model.

- *View pattern*, where the state of micro entities is computed to reflect the state of the macro entities they emanate from.
- *Cohabitation pattern*, where the interactions between micro and macro entities are bidirectional, so they influence each other.

B. Adaptive Resolution

In many multiscale frameworks, the level of detail of a sub-model is defined by its spatial or time resolution. In these scenarios it is necessary to specify the conditions that trigger a change of resolution. The *Adaptive Resolution* pattern involves the definition of these conditions, that is necessarily model-dependent. As an example, in [24] the authors propose an adaptive multiscale infection propagation model that combines the accuracy of ABMs with the computational efficiency of equations-based simulations. The model starts with the agent-based paradigm in order to thoroughly represent the initial dynamics of the diffusion of the pathogen, and then it switches to an equation-based methodology after a certain threshold of infected individuals is reached, so as to support a population-averaged approach.

VIII. CONCLUSIONS

In this paper we described a set of design patterns that can be used to address some of the issues encountered in the development of multilevel models. The patterns are divided into five categories. Orchestration Patterns are strategies to organize the various building blocks; Structural Patterns are design solutions for developing software systems with a hierarchical structure; Execution Policy patterns provide a rationale for executing the building block of a complex models; Multiscale patterns are solutions for representing a system with multiple scales of detail; finally, Information Exchange patterns describe how data can be exchanged among the sub-models.

The novelty of this proposal is to bring the methodological contribution provided by design patterns into the context of multilevel modeling and simulation. In fact, ad-hoc answers for recurrent modeling issues were missing from the state of art, as most of the effort in multilevel M&S has traditionally been devoted – with some exceptions – to application development rather than methodological studies.

The list of patterns described here is not exhaustive; we are currently working towards extending our collection by leveraging a recent review of the state of the art [1].

ACKNOWLEDGEMENTS

Moreno Marzolla was partially supported by the Istituto Nazionale di Alta Matematica “Francesco Severi” – Gruppo Nazionale per il Calcolo Scientifico (INdAM-GNCS) and by the ICSC National Research Centre for High Performance Computing, Big Data and Quantum Computing within the NextGenerationEU program.

REFERENCES

- [1] L. Serena, M. Marzolla, G. D’Angelo, and S. Ferretti, “A review of multilevel modeling and simulation for human mobility and behavior,” *Simulation Modelling Practice and Theory*, p. 102780, 2023.
- [2] A. M. Law, *Simulation modeling and analysis*. McGraw-hill, 2015.
- [3] H. Van Dyke Parunak, R. Savit, and R. L. Riolo, “Agent-based modeling vs. equation-based modeling: A case study and users’ guide,” in *Multi-Agent Systems and Agent-Based Simulation: First International Workshop, MABS’98, Paris, France, July 4-6, 1998. Proceedings I*. Springer, 1998, pp. 10–25.
- [4] S. De Marchi and S. E. Page, “Agent-based models,” *Annual Review of political science*, vol. 17, pp. 1–20, 2014.
- [5] S. Ghosh, “On the concept of dynamic multi-level simulation,” in *Proc 19th annual symposium on Simulation*, 1986, pp. 201–205.
- [6] L. Serena, M. Marzolla, G. D’Angelo, and S. Ferretti, “Multilevel modeling as a methodology for the simulation of human mobility,” in *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 2022, pp. 49–56.
- [7] P. D. Dans, J. Walther, H. Gómez, and M. Orozco, “Multiscale simulation of dna,” *Current opinion in structural biology*, vol. 37, pp. 29–45, 2016.
- [8] M. Martins, S. Ferreira Jr, and M. Vilela, “Multiscale models for biological systems,” *Current opinion in colloid & Interface Science*, vol. 15, no. 1-2, pp. 18–23, 2010.
- [9] J. Nguyen, S. T. Powers, N. Urquhart, T. Farrenkopf, and M. Guckert, “An overview of agent-based traffic simulators,” *Transportation research interdisciplinary perspectives*, vol. 12, p. 100486, 2021.
- [10] I. T. Haman, V. C. Kamla, S. Galland, and J. C. Kamgang, “Towards an multilevel agent-based model for traffic simulation,” *Procedia Computer Science*, vol. 109, pp. 887–892, 2017.
- [11] E. Cristiani, B. Piccoli, and A. Tosin, *Multiscale modeling of pedestrian dynamics*. Springer, 2014, vol. 12.
- [12] N. Mideo, S. Alizon, and T. Day, “Linking within-and between-host dynamics in the evolutionary epidemiology of infectious diseases,” *Trends in ecology & evolution*, vol. 23, no. 9, pp. 511–517, 2008.
- [13] A. E. S. Almocera, V. K. Nguyen, and E. A. Hernandez-Vargas, “Multiscale model within-host and between-host for viral infectious diseases,” *Journal of Mathematical Biology*, vol. 77, no. 4, pp. 1035–1057, 2018.
- [14] R. Qesmi, J. M. Heffernan, and J. Wu, “An immuno-epidemiological model with threshold delay: a study of the effects of multiple exposures to a pathogen,” *Journal of mathematical biology*, vol. 70, no. 1, pp. 343–366, 2015.
- [15] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.
- [16] M. Xiong, S. Tang, and D. Zhao, “A hybrid model for simulating crowd evacuation,” *New Generation Computing*, vol. 31, pp. 211–235, 2013.
- [17] G. D’Angelo, S. Ferretti, and V. Ghini, “Distributed hybrid simulation of the internet of things and smart territories,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4370, 2018.
- [18] D. R. Jefferson, “Virtual time,” *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, p. 404–425, jul 1985. [Online]. Available: <https://doi.org/10.1145/3916.3988>
- [19] P. A. Mboup, K. Konaté, and J. Le Fur, “A multi-world agent-based model working at several spatial and temporal scales for simulating complex geographic systems,” *Procedia Computer Science*, vol. 108, pp. 968–977, 2017.
- [20] S. R. Musse and D. Thalmann, “Hierarchical model for real time simulation of virtual human crowds,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 2, pp. 152–164, 2001.
- [21] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley, 2000.
- [22] B. Liskov and L. Shrira, “Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems,” *SIGPLAN Not.*, vol. 23, no. 7, p. 260–267, jun 1988.
- [23] P. Mathieu, G. Morvan, and S. Picault, “Multi-level agent-based simulations: Four design patterns,” *Simulation Modelling Practice and Theory*, vol. 83, pp. 51–64, 2018.
- [24] G. V. Bobashev, D. M. Goedecke, F. Yu, and J. M. Epstein, “A hybrid epidemic model: combining the advantages of agent-based and equation-based approaches,” in *2007 winter simulation conference*. IEEE, 2007, pp. 1532–1537.