What Good are Models?
Invited Paper

Edward A. Lee[1] and Marjan Sirjani[2,3]

[1] EECS Department, University of California at Berkeley, U.S.A
[2] School of Innovation, Design and Engineering, Mälardalen University, Sweden
[3] School of Computer Science, Reykjavik University, Iceland

Abstract. Models are central to engineering. They are used for analysis, synthesis, and communication between humans. A given artifact or process may have multiple models with different purposes, modeling different aspects, or modeling at varying levels of abstraction. In this paper, we give a general overview of how models are used, with the goal of making the concepts clearer for different communities. We focus on the domain of track-based flow management of automated systems, and identify two different modeling styles, Eulerian and Lagrangian. Eulerian models focus on regions of space, whereas Lagrangian models focus on entities moving through space. We discuss how the features of the system, like having centralized or decentralized control or the ability to install fixed infrastructure, influence the choice between these styles. Although the choice between modeling styles is rarely made consciously, it affects modeling efficiency, and one style may be far better suited for certain modeling problems than another. For problems with a more global nature concerning the physical space, an Eulerian model is likely to be a better match. For problems that concern the moving objects specifically, where the identity of the individual objects is important, a Lagrangian view is the one to choose. In many cases, combining the two styles is the most effective approach. We illustrate the two styles using an example of an automated quarry.

1 Introduction

We are now in the era of cyber-physical systems, the Internet of Things, and smart applications. For building such systems we need a team of experts with various domains of expertise, including computer science, software engineering, computer networking, control and communication engineering, robotics, and artificial intelligence. Although all these experts have the experience of working with models, they use different terminologies and very different modeling languages. Models are used differently in different contexts, and for people with varying backgrounds, this may create confusion. With the increasing need for people with different backgrounds to work together, communication is becoming a crucial obstacle. To tackle this obstacle, in this paper we provide an overall view of modeling in the contexts of science and engineering, with different

in Formal Aspects of Component Software, 15th International Conference, FACS 2018, Pohang, South Korea, October 10–12, 2018, Proceedings, Kyungmin Bae and Peter Csaba Olveczky, Eds., Springer, LNCS 11222.

goals of synthesis and analysis, and in different directions of abstraction and refinement. We show how validation and verification provide quality assurance on multiple levels in the process of modeling.

We focus on flow management of autonomous systems, covering a wide range of application domains including air traffic control, railway systems, road traffic, automated warehouses, smart transport hubs in cities, and computer networks. The increasing traffic volume that inevitably comes with increasingly efficient use of resources makes collision avoidance and safety assurance more critical and complicated. It also increases the possibility of unpredicted changes and demands automated runtime planning. Any improvement in planning can save a huge amount of cost, which may be in the form of time, fuel consumption, or risk, and can make a system more environmentally friendly and sustainable and improve user satisfaction.

Experimenting with the design of transportation systems in the field can be prohibitively expensive and unsafe. As a consequence, it is essential to use models both for understanding how existing systems work and for determining how to improve them. There is a surprising richness of possibilities for constructing models, and the choice of modeling strategy can strongly affect the outcome. What model to use depends not only on the problem domain, but also on the goal.

In the following sections, we give an overall view on modeling, abstraction and refinement, scientific and engineering models, verification and validation, and synthesis and analysis. We then continue by focusing on actor-based models of track-based flow management systems. We show how there are similar and common concepts, features, and problems in all flow management systems. We then present two views of flow management, Eulerian and Lagrangian, and offer a discussion on how to choose one of these models over the other and how and when to combine them.

We use an automated quarry as a running example of a track-based flow management system. The case study is inspired by the quarry site used in the electrified site project at Volvo Construction Equipment, where autonomous haulers (HX) are used for transporting material in the site, see Figure 1. HX machines are intended to perform tasks such as material transport, loading, unloading, and charging in a cyclic manner with predefined timing constraints and task priorities [1,2].

Note that there is a huge amount of work done on traffic flow management in various domains. The aim of this paper is not to serve as a literature review nor a comparative study of traffic management methods. The aim is to give an overall view of modeling of flow management systems from different perspectives.

This paper reflects the authors' collective experience in modeling different applications using actor-based modeling and simulation frameworks. Over decades of building models, we have found common patterns. Flow management problems consist of sources of moving objects, destinations, and paths. Models address safety and optimization goals, like collision avoidance, higher throughput, and minimum delay. Models also address policies for adapting to change. For

example, in a network on chip (NoC), we have to deal with a faulty router, and in air traffic control systems (ATC), we have to avoid storms. But perhaps the most interesting insight derived from our experience is that two very different and complementary modeling styles can be used for flow management problems. These two styles are called Eulerian and Lagrangian, after a similar dichotomy of modeling styles in fluid mechanics. Eulerian models focus on regions of space, whereas Lagrangian models focus on entities moving through space. When building actor-based models, an actor represents a region of space in an Eulerian model and a moving object in a Lagrangian model. Although the choice between modeling styles is rarely made consciously, it affects modeling efficiency, and one style may be far better suited for certain modeling problems than another.

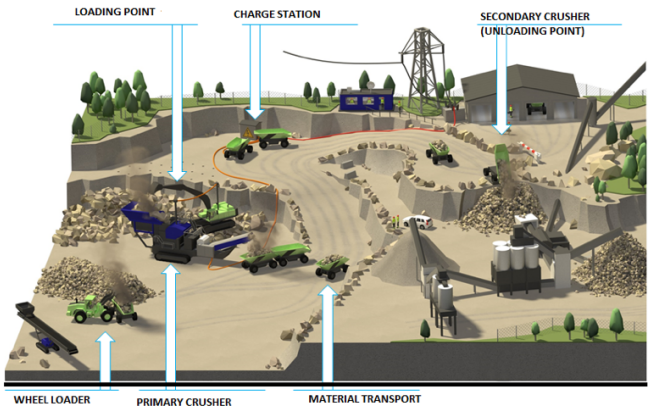


Fig. 1: The Volvo Quarry Site (from [1])

2 Modeling

A model is any description of a system that is not the thing-in-itself (*das ding an sich* in Kantian philosophy). By this definition, every human conception of a system external to herself is a model. In this paper, we focus on modeling a system that includes human-made artifacts and processes, not just naturally occurring systems. In other words, we are focused on engineering, not natural science.

In the automated quarry, the haulers and their controllers, for example, are human-made artifacts. But these artifacts are subject to natural laws, and therefore have properties and behaviors that are not human-made. Hence, our models will need to reflect both physical realities and design intent.

A naive approach to modeling such a system is to build a single ever-more-detailed model of its physicality. Such models quickly become unwieldy and incomprehensible. A more productive approach is to build a multiplicity of models, each with a purpose. But how should these models be built and how should they relate to one another? On what principles and languages should we base the models? We examine these questions in this section.

2.1 Abstraction and Refinement

Every model is designed to study some property or properties of a system. For example, we might be interested in the cost of a system, and we might construct a model where the cost is defined to be the sum of the costs of a collection of components. Here, the property of interest is cost, and this property lives within formal system of arithmetic where costs can be summed and compared.

A model A may be an **abstraction** of another model B in that they are intended to model the same thing, but A has less detail than B [3]. For example, B may be a model with three components with costs $c_1 = 10$, $c_2 = 20$, and $c_3 = 30$, so the total cost is 60. An abstraction A might be a simpler model with no components, but only a variable c, representing total cost, and an assertion that $c < 100$. A is an abstraction of B because it is intended to model cost, but it does not break down the cost by component, and it does not give a specific cost.

The abstraction A is **sound** if every property of interest that is true for A is also true for B (this is often called "property preservation"). We say "every property of interest" because any model will have properties that are not of interest. For example, when building model A, we are not interested in the number of components, so we abstract that property away. Only the total cost is a property of interest in A. For A to be a sound abstraction of B, it is necessary for total cost to also be a property of interest in B. When we use models, we focus only on some of the properties, and hence soundness is always with respect to these properties of interest.

When a model A is a sound abstraction of another model B, we can equivalently say that B is a **refinement** of A. A refinement B of A adds detail to A without exhibiting properties that are prohibited by A.

A concept that usually comes with **soundness** is **completeness**. A is a **complete** abstraction of B if every property of interest that is true for B is also true for A. While doing abstraction and refinement, we are rarely looking for completeness. Useful abstractions are usually sound but not complete. In our cost example, A is a sound abstraction of B, but if costs of individual components are also properties of interest for us, then A is not a complete abstraction, because in B we have the costs of individual components but in A we have lost that information.

There are cases where two different models exhibit the same behavior and satisfy the same properties. When building models of behavior using automata or transition systems, for example, the formal methods community uses the notions of simulation and bisimulation [4], and there are precise definitions for

each concept and the spectrum of different equivalency relations [5]. Abstraction and refinement are reduced to simulation relations, while bisimulation implies both soundness and completeness.

Note that both soundness and completeness are with respect to properties of interest. Suppose, for example, that we have a model C with three components with costs $c_1 = 10$, $c_2 = 20$, and $c_3 = 30$ and weights $w_1 = 1$, $w_2 = 2$, and $w_3 = 3$. If the "properties of interest" include only *costs*, then B is a sound and complete abstraction of C.

2.2 Scientific and Engineering Models

Following [6], we distinguish models that we call **scientific models**, which are intended to reflect the behavior of a *pre-existing system*, from models that we call **engineering models**, which are intended to specify the behavior of a *system to be built*. It is important to recognize whether a model is to be used in a scientific way or an engineering way. For example, adding detail may enhance a scientific model and degrade an engineering model. An architect probably should not specify the placement of every brick, but a structural engineer studying the earthquake safety of a building may need this detail.

An engineering model may serve as a **specification** for a system to be built. It may be informal or formal and more abstract or less abstract. The purpose of the model is to give properties that the built system is required to have. Engineering models are often layered, where a less detailed model A serves as a specification for another more detailed model B. For engineering models, **verification** is the process of checking that B is a refinement of its specification A, or, equivalently, that A is a sound abstraction of B.

Scientific models may also be more or less detailed and formal or informal. The most useful scientific models are the simplest (least detailed, more abstract) that still exhibit the properties of interest. A scientific model is **faithful** if every property it exhibits is also a property of the system being modeled. Faithfulness is similar to soundness, but while soundness is a relation between two models, faithfulness is a relation between a model and a thing-in-itself.

Faithfulness is easier to achieve if properties of the modeling language itself reflect properties of the problem domain being modeled [7]. For example, a modeling language with continuous time will make it easier to faithfully model a system with continuous dynamics. Similarly, a modeling language with discrete events, concurrency, and asynchronous interactions will make it easier to model distributed software systems.

Scientific models may also be layered, although this is far less common than for engineering models [6]. If model A is a sound abstraction of model B, and model B is faithful to some physical system C, then A is also faithful to C.

Faithfulness is much harder to pin down formally than soundness because it is not a relationship between models. It is a relationship between a model and physical, real-world system, the thing-in-itself. Any property held by a real-world system is ultimately subject to measurement error, and hence faithfulness can never be more than approximately assured. Reflecting this fact, Box and

Draper famously said, "all models are wrong, but some are useful" [8]. They were referring to scientific models, not engineering models. A specification, an engineering model, is by definition right, not wrong.

For most properties of interest, in science, models are always wrong, in the sense of Box and Draper, but in engineering, a physical, real-world implementation is always wrong. The specification is an idealization, and an implementation can only approximate it. For instance, every physical realization is vulnerable to failures that are not accounted for in the specification. How will a computer program behave, for example, if the computer it is running on is immersed in salt water? Whatever behavior emerges is likely not a behavior of the specification.

Consider for example the dynamic behavior of an electric hauler reacting to network command to accelerate. A scientific model may use Newton's laws to describe how the vehicle reacts to torque induced by the motor. This model is wrong, in the sense of Box and Draper, because it depends on parameters, such as the weight of the vehicle, that cannot be perfectly measured. It is also wrong because Newton's laws are wrong in that they fail to account for relativistic effects. But with appropriate assumptions, the model remains useful.

On the other hand, an engineering model for the same problem is a specification. It defines the correct behavior of a hauler being designed. But no physical vehicle will perfectly match that behavior, and therefore the real-world physical implementation is wrong. But mirroring the usefulness of a scientific model, with appropriate assumptions, the physical implementation will be useful.

2.3 Verification and Validation

According to Boehm [9], **verification** is to establish the truth of the correspondence between a software product and its specification, and **validation** is to establish the fitness or worth of a software product for its operational mission. Boehm refers to the roots of the words as well. "Verification" is derived from the Latin word for "truth", *veritasis*, and "validation" is derived from the Latin word for "to be worthy", *valere*. Informally, we might define these terms by asking "am I building the product right?" (verification) and "am I building the right product?" (validation).

Validation is comparing a model with the system, or to be more precise, comparing a model with the system projected over behaviors of interest. The model defines the "right product." Verification is comparing the model with another model reflecting more abstract properties. To avoid sinking into a philosophical quagmire, we can only formally establish "truth" by comparing models.

For engineering models, verification means making sure that a model B exhibits only acceptable behaviors, or, equivalently, that it does not exhibit prohibited behaviors. A specification A is a reference point that defines acceptable behaviors and their complement, prohibited behaviors. To formally verify B is to prove that it is a refinement of A.

Validation in an engineering process means to check whether the *specification* is written correctly, i.e. whether the model you built as the specification is really

representing the system you want to be built eventually. Validation is checking whether the product meets the customer expectations and requirements.

For scientific models, validation is checking how much the model reflects the existing system being modeled, or, equivalently, how **faithful** the model is. Here, scientists rely on the scientific method to approximately validate models. Specifically, they design experiments that have the potential to expose mismatches between the behavior of the model and that of the thing-in-itself. That is, the experiment has the potential to **falsify** the model. Failure to falsify a model is the best available validation of a scientific model. On the other hand, for verification, you need two models to compare. For verification of both scientific and engineering models, you assume that the specification (which is as a reference model) is correct (valid) and verify that the other models that you build based on that are refinements.

Note that no scientific model is perfectly faithful to any physical system unless it is the physical system itself. Hence, it is not necessarily a mistake to fail to reflect behaviors of the system being modeled. All scientific models do that, in that some behaviors are ignored or abstracted away. It is a mistake to fail to reflect *behaviors of interest*, behaviors that the model was intended to explore. A scientific model can therefore be viewed as projection of a system onto a subspace of behaviors of interest.

For example in our automated Quarry, the customer has a safety and progress *requirement* that if the hauler faces an unpredicted obstacle, it has to avoid the obstacle (safety), but the system should not completely shut down (liveness). This requirement is then formulated mathematically, for example as a set of temporal logic formulas. The formulas must be written in a concrete way, for example in terms of the data received by the hauler from its sensors and cameras and commands issued to its actuators. For example, a temporal logic formula may specify that the machine halts if an obstacle is sensed, and that this halting state is temporary. Checking whether this formula is correctly capturing the customer requirements is a validation process. The formula is now a *specification* of the system. When the controller program of the hauler is being developed, the behavior of the hauler executing this program is verified against this specification.

2.4 Synthesis and Analysis

Models can be used for both synthesis and analysis. In a model-driven development approach we do **synthesis**; that is, we build abstract models that serve as a specification of a system to be built, and then we refine the models, adding details until we build the system itself. Typically, the process is iterative, with the specifications evolving along with their refinements. Models can be used along the way for different **analysis** purposes, verification, validation, and performance evaluation. If we have formal and automatic refinement techniques, we may be able to avoid introducing errors in the refined models while details are added. In this case, synthesis is said to be "correct by construction."

A classic example of correct-by-construction synthesis is what a compiler is intended to do. It translates a specification (a program) into a refinement (machine code for a particular machine), adding detail about how to accomplish the intent of the program while preserving behaviors specified by the program. If this is done correctly, the machine code will not exhibit any behaviors that are prohibited by the program. Note that the machine code is still not an implementation. It is another model, specifying behaviors that a silicon implementation of a processor is expected to exhibit. Since it is a model, not a thing-in-itself, the machine code can be formally verified, proven to not exhibit behaviors prohibited by the program. The thing-in-itself, of course, will always be able to exhibit behaviors prohibited by the program, if it is immersed in salt water for example.

Compilers that can be trusted to produce correct machine code have proven to be a spectacularly powerful engineering tool. Spurred in part by this success, software engineers continue to try to push up the level of abstraction at which systems are specified and develop correct-by-construction synthesis techniques that work from those more abstract models. These efforts have met with limited success. A commonly used alternative to correct-by-construction synthesis is to treat a model, such as a program, as a pre-existing artifact, and to construct an abstraction, a scientific model of the program. This model can be used for analysis. In some cases, the abstract model can be constructed automatically, using for example abstract interpretation [10]. We could call such a process "correct-by-construction abstraction."

For example, instead of synthesizing a computer program from a more abstract specification, say in UML, we may write a program by hand and build an abstract model of that program to analyze its behaviors. The more abstract model is, effectively, a scientific model of an engineering model. For example, a nondeterministic automaton could model a computer program. We can then perform model checking [11], which formally checks that the automaton is a refinement of a still more abstract specification model, given for example as a set of temporal logic formulas. If the automaton is a sound abstraction of the program (ideally, it is because it was built using correct-by-construction abstraction), and the automaton is a refinement of the specification (checked using model checking), then the program is a refinement of the specification.

Model checking, simulation, and building physical prototypes can all be used as methods for analysis. Simulation, which is the execution of an executable model, reveals one possible behavior of a model with one set of inputs. Model checking reveals all possible behaviors of a model over a family of inputs.

Different communities may prefer one technique over others. Some practitioners, for example, prefer physical prototypes over simulation, saying that "simulation is doomed to succeed." Rodney Brooks, for example, writing about robotics, says "there is a vast difference (which is not appreciated by people who have not used real robots) between simulated robots and physical robots and their dynamics of interaction with the environment" [12].

Indeed, simulation can be misused. A simulation of a robot may be the execution of an engineering model, a specification. If the specification is valid, then

the simulation is indeed doomed to succeed. The model should not be misinterpreted as a scientific model that reveals unknown or unexpected behaviors of the thing-in-itself.

When using simulation, it is important to understand whether one is doing engineering or science. An engineering model should not be used to discover how a real physical system will behave because it will only reveal behaviors that were designed in. Faithful scientific models of robots are indeed difficult to construct because robots exhibit complex physical behaviors that are affected by phenomena such friction, plastic deformation, and acoustic propagation of vibration that are notoriously difficult to model [13]. A good engineering model of a robot, however, can be useful for validation of a specification. Does the specification, an idealized robot, exhibit desired behaviors? It becomes a separate question whether a real robot, a thing-in-itself, can be built so that the specification model is faithful.

When faithful scientific models are not available, physical prototypes are used. Physical prototypes will reveal problems that simulation based on an engineering model cannot reveal. A robot arm, for example, may be modeled as rigid and frictionless for the purposes of developing path planning algorithms. A hauler in an automated quarry may be modeled as moving at a constant speed or stopped (two states) if the purpose of the model is to analyze congestion and optimize throughput. These models should not be used to analyze precision of motion.

3 Actors

A component is a chunk of functionality that can be composed with other chunks of functionality to yield a new chunk of functionality. In software engineering, different classes of component models have evolved. In imperative languages, for example, a component is a procedure, and a program is a sequential execution of a top-level procedure that can call other procedures. Components are composed by procedure calls, a temporary transfer of the flow of control from one procedure to another. Object-oriented languages are organizations of imperative programs with information hiding. In functional languages, a component is a stateless function (free of side effects), and components are composed by function composition. In actor languages, components are concurrently executing programs called "**actors**" that send messages to one another over streaming channels. Actor languages have proved very effective for modeling concurrent and distributed systems, so we focus on those here.

The term "actor" was introduced in the 1970's by Hewitt to describe the concept of autonomous reasoning agents [14]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [15]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. Each actor has a single input queue on which it receives messages, and it handles messages in order of their arrival. Rebeca [16,17], for example, is a software framework that realizes Agha's actors, match-

ing asynchronous and event-driven domains. It has proven particularly suitable for modeling and analyzing network protocols and applications [18,19,20].

The term "actor" has also been used for **dataflow** models of computation. Three major variants of dataflow models have emerged in the literature: Kahn process networks [21], Dennis dataflow [22], and dataflow synchronous languages [23]. In all three, as with Hewitt actors, a program is a network of interconnected actors. Unlike Hewitt actors, dataflow actors have explicit input and output ports, and rather than referencing a remote actor to send a message to it, dataflow actors send messages to output ports and the network handles routing that message to one or more destinations. Since actors do not have references to one another, dataflow actors tend to be more modular and reusable than Hewitt actors. The same actor can be instantiated in multiple contexts.

In Dennis dataflow, program execution is a sequence of atomic actor **firings**, where each firing consumes input **tokens** (chunks of data) and produces output tokens. In Kahn networks, each actor is a sequential program that reads from input ports and writes to output ports. In the original Kahn-MacQueen variant [24], a read from an input port will block until an input token is available and writes to output ports are nonblocking, sending data with no constraints. Various generalizations allow richer input-output semantics, for example to allow for nondeterministic merging of streams, and various specializations constrain the execution, for example to prevent unbounded buildup of unconsumed tokens in queues. Dennis dataflow can be viewed as a special case of Kahn process networks [25]. Dataflow synchronous languages differ from both of these in that, semantically, all actors in a program fire simultaneously and instantaneously. The inputs and outputs of the actors are defined by a fixed point of the function defined by the composition of actors.

The Ptolemy II framework [26,27] generalizes actors to embrace any model of computation (**MoC**) where a program is either a static or dynamic graph of components with ports, where the components are concurrent, and where the ports are connected through communication channels. In a Ptolemy II model, the execution of such a graph is governed by a **director**, a coordinator that realizes the specific MoC. Directors have been realized for Kahn process networks, several flavors of dataflow, and dataflow synchronous models. In addition, directors are available for MoCs that do not traditionally fall within the purview of actor models, but which share essential features with actor models. These include a discrete-event (DE) model, where communication is via time-stamped events, a rendezvous MoC, where concurrent components communicate by rendezvous, and a continuous-time MoC, where the communications between components are semantically continuous-time signals. Ptolemy II DE models are similar to many simulation frameworks such as DEVS [28] and hardware description languages such as Verilog and VHDL. Ptolemy II rendezvous models are similar to Reo [29] and realize a semantics similar to communicating sequential processes (CSP) [30]. The continuous-time models of Ptolemy II are similar to those in modeling languages such as Simulink (from The MathWorks) and Modelica [31].

A key innovation in Ptolemy II is that many of these MoCs can be hierarchically combined in the same model by leveraging an actor abstract semantics [32].

In any framework that supports composition of communicating actors, the specific semantics of the interaction between actors, the MoC, is a meta-model, a model of a family of models [33]. The MoC is an essential part of any modeling language. It provides designers with constructs and features to build programs and models, and the semantics of the meta-model shape the models that are built, sometimes without the designer realizing that this is happening.

Design patterns and templates also function as meta-models, using constructs and features that are provided by the modeling language and adding guidelines for how to model. They tell designers how to match entities in the problem domain to entities in the solution domain (the model we are building). Design patterns can shape the thoughts of the designer.

Broadly, these varied actor languages, semantics, and modeling frameworks provide us with constructs and features that fit concurrent and distributed systems. The varying semantics are tuned for different problem domains. In this paper, we examine how some of the relevant MoCs fit track-based flow management of automated systems.

4 Eulerian and Lagrangian Models of Track-based Systems

We now focus on the track-based flow management systems, specifically traffic management systems and transportation. By "track-based" we mean that movement through the space is restricted to pre-defined paths, as opposed to unrestricted movement in two or three-dimensional Euclidean space. Air traffic control, railroad scheduling, unmanned aerial vehicles (UAV) traffic management, smart transport hubs in cities, automated warehouses, and autonomous transport vehicles (ATVs) are examples where we have track-based traffic and transportation. Wired computer networks, like networks on chip (NoCs), demonstrate similar patterns of features, behavior and goals.

Different models and techniques are used in different application domains for flow management of such systems. The main concerns are guaranteeing safety (like avoiding collisions or running out of fuel) and improving efficiency (including multi-objective optimizations like reducing delays, maximizing throughput, decreasing fuel consumption, and minimizing environmental impact).

Design patterns help when dealing with similar problems by providing a template as the basis for designing your solution. You can reuse, customize and optimize similar techniques. We distinguish two general patterns in building models for flow management. The first pattern, called "Lagrangian," focuses on the moving objects, such as airplanes, trains, automated vehicles, commuters in cities, robots and products in warehouses, and packets in NoCs. In the Lagrangian view, the moving objects have independent identities. The properties of interest concern the behaviors of individual moving objects, including for example how quickly and safely they reach their destinations.

The second view, called "Eulerian," focuses on a region of space, and models the aggregate traffic passing through the space. In the Eulerian view, each region of space, such as a track or a section of a track, has an independent identity, and the objects moving through space are anonymous, possibly even indistinguishable from one another. The properties of interest concern the utilization of space, including for example congestion and throughput.

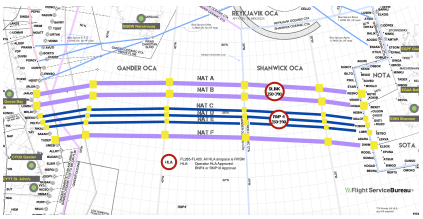
The Eulerian-Lagrangian terminology comes from fluid Mechanics and classical field theory, where Lagrangian and Eulerian models are well understood as alternative ways to model fluid flow [34]. In a Lagrangian model of a fluid, the observer follows an individual fluid parcel as it moves through space and time. In the Eulerian view, the observer fixes on a region of space and observes fluid mass passing through that space. For example, in studying the flow of a river, the Lagrangian view can be visualized as sitting in a boat and floating down the river, whereas the Eulerian view can be visualized as sitting on the bank and watching the boats float by.

4.1 Flow Management: A Generic View

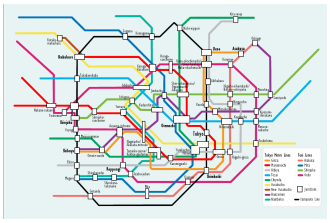
In a track-based flow management system, moving objects are constrained to follow one-dimensional tracks within a two or three-dimensional space. Tracks can be joined and split at intersections. At any given time, a track-based system will define a network of interconnected tracks and each track will be occupied by some number of moving tokens. Sources and sinks for tokens represent the edges of the network being modeled.

The nodes in the network represent sources, destinations, and intermediate destinations. Sources and destinations can be airports (in ATC), train stations (in a railway system), hubs (in smart transport hubs in cities), shelves or racks (in a warehouse), loading or unloading positions (in a quarry), or routers in NoCs. Intermediate destinations can be places that the moving objects may or must stop, like connecting airports for airplanes or charging stations for automated vehicles. The edges in the network represent tracks and sub-tracks. There can be a capacity assigned for each (sub-)track, and minimum and maximum allowed speed. In addition to Figure 1, Figure 2 shows applications where we can see the track-based flow management pattern. Figure 2.a shows the North Atlantic Organized Tracks (NAT-OTS) that is used for track-based air traffic control. Figure 2.b shows the subway map of Tokyo. Figure 2.c shows a small example of smart transport hubs in a city and how the commuters have choices for moving between these hubs [35]. Figure 2.d shows an array of routers on an NoC architecture similar to what the authors used in [18].

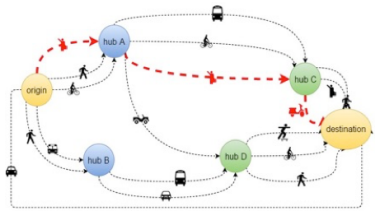
Moving objects form the flow on the network. Objects may have some attributes assigned to them. Apart from an identifier, they may have a maximum and an optimum speed, capacity for fuel, fuel consumption rate as a function of speed, node of origin, target node, and a pre-specified route. The model may go further than that and see each moving object as a smart agent with beliefs, goals, and learning capabilities.



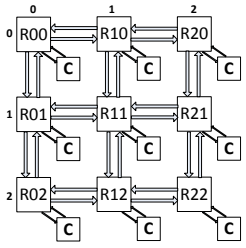
(a) By Coisabh [CC BY-SA 4.0] from Wikimedia Commons, https://commons.wikimedia.org/wiki/File:NAT-Tracks-24FEB17.png



(b) From Wikimedia Commons, https://commons.wikimedia.org/wiki/File:Tokyo_subway_map.PNG



(c) An abstract schema of smart transport hubs (courtesy of Jacopo de Berardinis, Carlo Castagnari, Giorgio Forcina from a presentation prepared on the work in [35]).



(d) An abstract schema of Network on Chip (courtesy of Mahdi Mossafa, after [18]).

Fig. 2: Some applications with the common attribute of flow management.

Many of the problems can be formulated as an **optimization** problem, where the **goals** generally are increasing efficiency of the traffic system, enhancing mobility (which means increasing throughput), reducing delay, or minimizing cost (like fuel consumption or environmental costs). The **constraints** mostly concern safety, like keeping the necessary separation between vehicles or limiting the number of moving objects in a track.

Another common problem in flow management is **adapting to change**. Airplanes get delayed and schedules have to be updated, weather conditions change, requiring rerouting, and new flights are added, requiring re-planning. Similar changes occur in railroads, transport hubs, warehouses, and other flow management applications, and consequently, such systems have to be adaptive and resilient to change.

An Eulerian model focuses on each track, the configuration and the connecting network of tracks in a more macroscopic way, while a Lagrangian model focuses on moving objects and their behavior in a more microscopic way. The decision to use an Eulerian or Lagrangian pattern, or even a combination, can have a profound effect on the effectiveness of a model. In our experience, practitioners rarely make this decision consciously. They just build the first model that pops into their heads.

4.2 Eulerian and Lagrangian Actor Templates

In an actor-based model we can realize Eulerian and Lagrangian views using track-as-actor or moving-object-as-actor, respectively. In a track-as-actor pattern (Eulerian), tracks are modeled as actors, and the moving objects are modeled as messages passing between actors. In a moving-object-as-actor pattern (Lagrangian), each moving object is modeled as an actor, and actors have some local information from their surroundings and the configuration of the system and may be able to autonomously decide their next move.

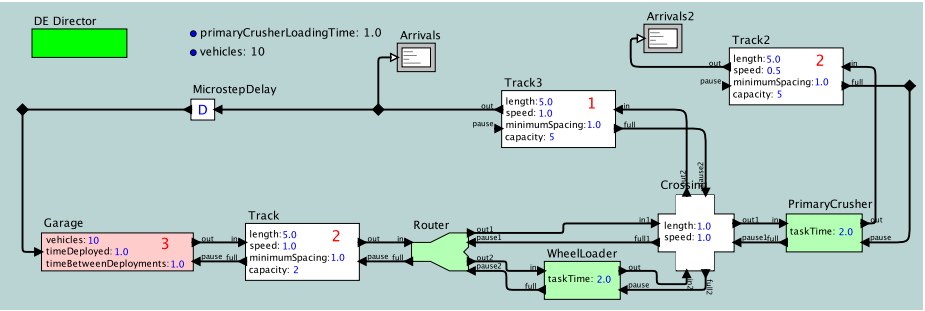


Fig. 3: Eulerian model of a piece of the automated quarry of Figure 1 built in Ptolemy II.

Figure 3 shows an Eulerian model of a section of the automated quarry rendered in Figure 1. This model is built using Ptolemy II [27] with a discrete-event director. The actors are shown as boxes with input and output ports through which time-stamped messages are sent. Each actor reacts to messages in time-stamp order.

It is a matter of choice to build an Eulerian model and to use a discrete-event model of computation. What are the consequences of these choices? Most obviously, the active agents in the model are tracks, garages, and workstations such as a wheel loader and crusher. The vehicles, which are electric haulers in this model, are represented by messages exchanged between actors. This model does not track individual vehicles, but rather models queuing and congestion at locations in the quarry. As a consequence, this model could be suitable for planning routing strategies, but is probably not suitable for developing or evaluating collision avoidance or battery management strategies. Note that one way to handle collision avoidance in this model is to have tracks with a capacity of one where the physical space is (virtually) divided in to tracks with a length that represents the minimum safe separation between the moving objects. This may not be a practical design, especially where we have a large space and small safe separation, because it will create a very fine-grain model with huge number of tracks. Also, it does not really model separation between individual vehicles, since a vehicle at the end of one track can be arbitrarily close to a vehicle at the beginning of the next track.

One consequence of choosing a discrete-event (DE) MoC is that the model is, by default, deterministic [36]. Building a nondeterministic model requires inserting actors with explicitly nondeterministic behavior. To model probabilistic events, the modeler has to explicitly insert sources of stochastic events, typically driven by pseudo-random number generators. In contrast, if we had chosen a Hewitt actor model, then the modeling framework itself would be nondeterministic in that messages arriving at an actor from distinct source actors are handled in nondeterministic order. In DE, the time stamps determine the order in which events are handled, and the time stamps are computed deterministically in the model.

To understand the consequences of the determinism of the modeling framework, consider the Crossing actor at the lower right in Figure 3. This represents an intersection where haulers going left to right cross haulers going bottom to top. How should we model the situation where two haulers arrive simultaneously at the intersection? In a DE model, simultaneous arrivals is a well-defined concept (the messages have the same time stamp), and how the model handles the situation has to be explicitly specified. In the particular model shown in the figure, we chose to handle it by giving priority to the haulers traveling from left to right, but we could equally well have handled it with a Bernoulli random variable (a pseudo-random coin flip). We could also have modeled it nondeterministically by inserting into the model an actor with an explicitly nondeterministic state machine, like that shown in Figure 4. The red transitions are

both enabled when messages with the same stamp arrive on inputs `in1` and `in2`, and which transition is taken will determine which message is handled first.

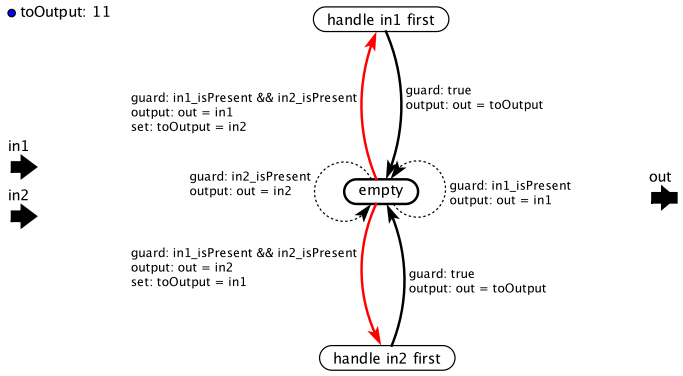


Fig. 4: Nondeterministic state machine actor built in Ptolemy II that merges streams with simultaneous messages nondeterministically into one stream.

If we wish to model the crossing nondeterministically, then a Hewitt actor model could be suitable. However, the original Hewitt actor model is untimed. To explicitly include timed behavior and nondeterministic ordering of simultaneous messages, we can use Timed Rebeca [37,38], which extends Hewitt actors with time stamps. Similar to the DE model, simultaneous arrivals is a well-defined concept (the messages have the same time stamp), but unlike the DE model the order of handling of messages with the same time stamp is always nondeterministic. Timed Rebeca has a built-in model checker that can be used to explore all the behaviors allowed by this nondeterministic semantics.

For some modeling problems, a nondeterministic MoC may be preferred because it can concisely express a family of possible behaviors that can be analyzed by exhaustive techniques such as model checking. Representing nondeterministic behavior is especially desirable in situations where we have for example simultaneous arrival of messages (vehicles, packets, ...) and no explicit priority-based policy to choose one over the other, so, the behavior of the thing-in-itself is really unknown to us. Then we build a nondeterministic model as a scientific model that shows different possible behaviors; this model will be used for analysis purposes, more specifically for model checking.

Such nondeterministic models, however, should not be used for simulation, because simulation reveals only one of many possible behaviors at a time. Unless the simulation resolves the nondeterminism in the same way that the system itself does, its choice of behaviors will tell us nothing about the likelihood of any particular behavior manifesting in the deployed system. Most implementations of Hewitt actors, for example, handle messages in order of arrival, but unless the system being modeled is actually a set of concurrent processes running on the

same multitasking scheduler as the model, the fact that a particular outcome occurs in simulation would say nothing about whether that outcome will occur in the field. A nondeterministic model is about what may possibly occur in the field, not about what is likely to occur. To assess the *likelihood* of particular outcomes, an explicit stochastic model with specified probability density functions should be built, not a merely nondeterministic model.

Deterministic models are often preferable for synthesis, where we want the system to behave as our specification. Allowing nondeterminism in the behavior of the model of the system is introducing an unnecessary risk in the design process (it's different from allowing nondeterminism in the model of the environment for analysis purposes). There are occasions that we may allow nondeterministic models as our specification for synthesis. If we are able to prove the properties of interest for the nondeterministic model (basically by definition properties of interest must hold for the specification), then the model can be used for synthesis. For example the model specify what safe behaviors are, and a synthesis algorithm optimize the design by choosing the best among the nondeterministic behaviors.

A more subtle consequence of the choice of the DE modeling framework shows up as bidirectional links between components. Notice that any actor that represents a place with finite capacity, which in the model is all actors except the Garage, has to apply back pressure to any actor that sends vehicles to it. When it is full, it notifies the upstream source by sending a "full" event, which has the effect of causing vehicles to wait in a queue upstream.

An alternative MoC that would not require backwards-flowing "full" messages would be based on a rendezvous-style of concurrency, using for example the Rendezvous director in Ptolemy II or a framework like Reo [29], which realizes a communicating-sequential processes (CSP) MoC [30]. In such an MoC, a sender of a message cannot complete a send until the receiver of the message is ready to receive it. Hence, back pressure is built in to the MoC and does not have to be explicitly built in to the model. However, no mature implementation that we know of a rendezvous-based modeling framework also models timed events, despite the fact that timed process algebras have been a subject of study for a long time [39]. Without a model of time, it would be difficult to use the model to optimize throughput, for example.

For some track-based applications, such as air-traffic control, back pressure that causes upstream queuing can be problematic. Haulers on the ground can stop and wait, but airplanes cannot. A track-based DE model of an ATC scenario is given in [40], but in that implementation, the DE director was subclassed to create a global coordinator that manages the back pressure. In effect, the MoC was modified to support the application domain.

A simple Lagrangian model for the automated quarry of Figure 1 is shown in Figure 5. This hierarchical model is very different from the Eulerian model of Figure 3 and could be used to answer a different set of questions. The scenario it models is two automated vehicles in a track, one leading and one following, with sensor data in the following vehicle giving it an estimate of its distance to

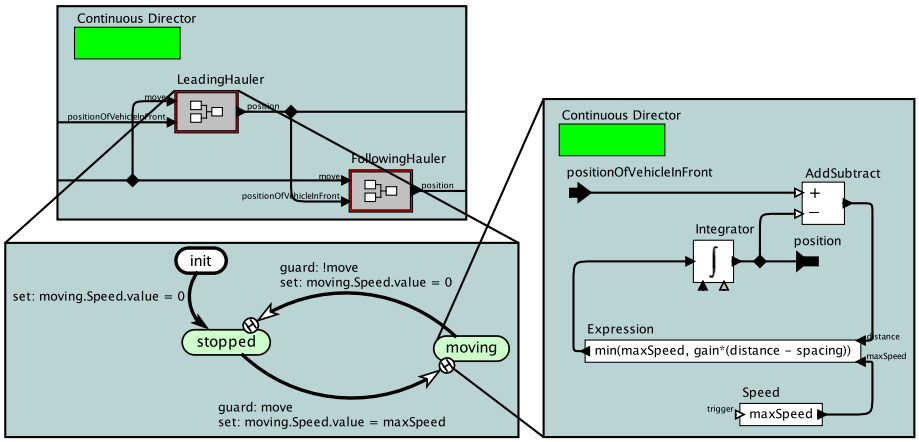


Fig. 5: Lagrangian model of a piece of the automated quarry of Figure 1 built in Ptolemy II.

the leading vehicle. The two actors at the top level of the model each represent a vehicle. Each actor is an instance of a class that, in this simplified model, is a hybrid system with two modes of operation, stopped and moving. In the moving mode, a first-order ordinary differential equation (ODE) models a proportional controller that strives to maintain a fixed distance to the leading vehicle but with a speed limit. This model could easily be elaborated with vehicle dynamics representing its loaded and unloaded inertia, for example, or its battery state as a function of time.

The model in Figure 5 is a continuous-time model, using the Continuous director in Ptolemy II [41]. It could equally well have been constructed in Simulink/Stateflow, a widely used modeling environment from The MathWorks. In the continuous MoC, the messages exchanged between actors represent continuous-time signals as approximated by an ODE solver.

Comparing the Eulerian and Lagrangian models is instructive. They are very different from one another. The Eulerian model could be used to evaluate overall throughput and to develop routing and path planning strategies for haulers. The Lagrangian model could be used to design the control system for the haulers that maintains a safe distance between them and to evaluate its performance.

4.3 Eulerian and Lagrangian Actor Models in Practice

It is relatively rare, when building models, to explicitly and consciously evaluate alternative modeling styles and choose one. A notable exception is documented by Menon, Sweriduk, and Bilimoria, who advocate Eulerian models for air traffic control [42]. Although it is common to build Eulerian models in this problem domain, it is rare to see this choice explicitly defended. Menon et al. observe that the complexity of the resulting model is reduced because it depends only

on the number of spatial control volumes and not on the number of aircraft. They show that the simplified model admits the use of linear control theory for analysis and design of flow control strategies. Menon et al. credit a number of authors dating back to 1955 for Eulerian approaches to modeling road traffic, which is apparently where they got the inspiration to apply the approach to air traffic control (it is worth noting that the use of Eulerian approaches in fluid mechanics goes back much further).

We encounter the two views of Eulerian and Lagrangian models by working on actor-based modeling of different distributed applications with the common pattern of flow management. Using an actor-based model, and seeing actors from an object-oriented point of view, we generally map each active entity in the system to an actor in our model. In packet routing applications on network-on-chips (NoCs), a router is an active entity, a piece of hardware and software that routes the packets through wires; and packets are passive entities that enter the network, and are routed along until they get to their destination. A natural mapping is modeling routers as actors, and packets as messages, which matches an Eulerian model. This Eulerian model represents a natural and faithful model of the system, and captures all properties of interest in order to answer the safety, optimization, and adaptation questions.

An actor-based Eulerian model of a network-on-chip is given by Sharifi et al. [18], who check efficiency and possibility of deadlock for different routing algorithms. A schematic of the NoC architecture they have analyzed is given in Figure 2.d. Ptolemy II, which we used to build the models in Figures 3 and 5, has also been used for Eulerian network simulation [43]. The widely used network simulator ns-3 (`https://www.nsnam.org/`) models networks in an Eulerian way using discrete-event semantics similar to that in Figure 3.

A seemingly different application is air traffic control (ATC). In air traffic control problems, tracks and sectors are an artifice. The space through which aircraft fly is continuous, not discretely divided into chunks. Moreover, aircraft themselves can be considered autonomous entities because pilots can override the commands received from ATC. Hence it may seem more natural to model ATC in a Lagrangian way, so that the individual entities in the model match the individual entities in the thing-in-itself. But with a more careful look, we see that a Lagrangian model may be overkill. In ATC systems we have centralized supervisory control, with a global and macroscopic view of the flow. Moreover, this macroscopic view is sufficient for analyzing optimization questions like minimizing delay and maximizing throughput. The macroscopic view is also sufficient to cover properties of interest such as collision avoidance. In an Eulerian model, collisions can be avoided in model with (sub-)tracks with capacity one; each (sub-)track becomes like a critical section where we enforce mutual exclusion. An alternative strategy is to consider a maximum load (of greater than one) for each track but leave the assurance of the collision avoidance to another Lagrangian model (as done in our Ptolemy example in Figure 3).

Some properties which may seem to need a Lagrangian view, like preventing an aircraft from running out of fuel, can be handled using an Eulerian model by

adding a few attributes to the messages (representing the aircraft). An Eulerian view works more efficiently, where we map tracks to actors, and airplanes to messages.

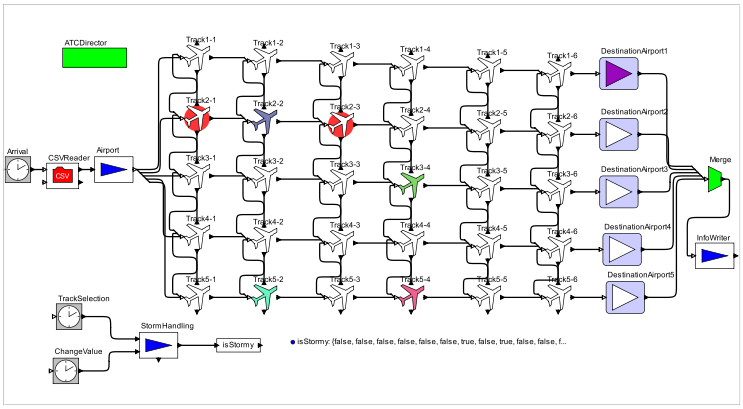


Fig. 6: Ptolemy model of the air traffic control example (courtesy of Maryam Bagheri)

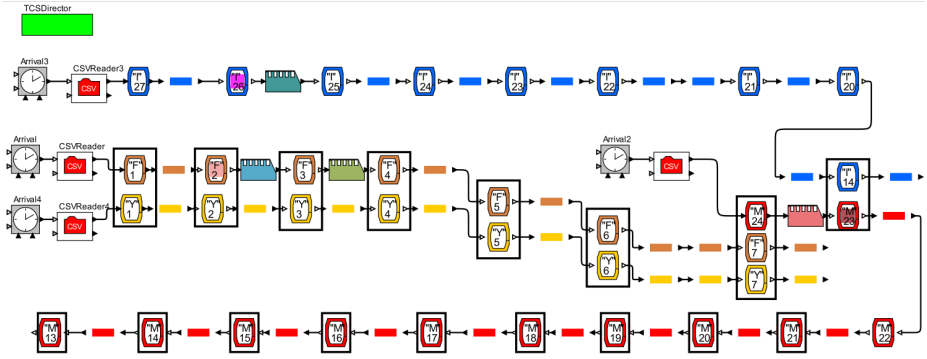


Fig. 7: Ptolemy model of the subway example (courtesy of Maryam Bagheri)

Bagheri et al. show that an Eulerian actor model is effective for designing self-adaptive track-based traffic control systems for air and railway traffic [40]. Their models are used for re-planning in the event of disruptions due to weather or other causes. Figures 2.a and 2.b show the two application domains tackled in [40]. An abstract view of the Ptolemy models of a simplified version of the above applications are shown in Figures 6 and 7.

A formal actor-based Lagrangian model of a smart airport with autonomous transport vehicles (ATVs) is proposed by Khakpour et al. in [44]. Each transportation service in the airport is realized by ATVs that transport passengers between stopovers, including passenger entrances, check-in desks, and departure gates. ATVs are modeled as Rebeca actors, and adaptation is handled by policies that govern the change of configuration and role of each ATV. Analysis techniques are provided to check the safety properties of the model. They added Eulerian models to their case study to provide a balance between centralized and decentralized control, a so-called controlled autonomy. They partition the smart airport area into smaller regions called cells. A cell contains an autonomous cell controller, deployed within the cell physically. A cell controller is aware of the ATVs and other subsystems located in its defined physical area, and provides nonlocal information for ATVs. This is an example of how we have a Lagrangian microscopic view on the autonomous machines, modeling their behaviors and their different configurations; and have an Eulerian macroscopic view on the physical space, modeling the state and configuration of each cell, and use both views to have a better control over the system on different levels.

Jafari et al. give an Eulerian model for flow management of the automated Volvo Quarry with electric self-driving machines, shown in Figure 1 [2]. The model is built in Timed Rebeca, as a scientific model for analysis purposes. The model checking tool in Rebeca is used for safety checking, it is also used for better flow planning using a reachability-based approach. Complementing the Eulerian model, Jayanthi Surendran Nair build a more detailed Lagrangian model for the Volvo Quarry site, with a focus on the behavior of each machine and the machine's architecture [45]. This model is used as an engineering model to be the basis for building the controller of each automated machine.

Castagnari et al. give an Eulerian macroscopic view of mobility services in a city with a network of transport hubs [35]. Figure 2.c shows a simple schema of multiple smart hubs, the commuters, and different mobility services that they can use, like a bus, bike, or taxi. The model is used as a scientific model for efficiency and mobility analysis and can work as lightweight preprocessing to prepare for microscopic simulations where commuters are modeled as agents in a Lagrangian style [46].

4.4 Eulerian Versus Lagrangian Models

In the following, we discuss some criteria for choosing to use an Eulerian or a Lagrangian model.

Modeling efficiency. When the individual identity of the moving objects are not important, then an Eulerian model is often the better choice because it avoids unnecessary detail. For example, in [47], Bayen et al. apply the idea from Menon et al. to show that an Eulerian model is effective at predicting aircraft counts in congested airspace. Elaborating this work, Sun et al. evaluate three Eulerian models for air traffic control [48]. In ATC, a "sector" is a portion of the airspace controlled by a single human air traffic controller. According to Sun

et al., traffic flow management includes maintaining the aircraft count in each sector below a legal threshold in order to ease the controller's workload as well as to ensure the safety of the flights. They say that this task is quite cumbersome and that extensive traffic forecast simulations that include all airborne aircraft are computationally too expensive. They advocate the use of Eulerian models instead.

For some problems, on the other hand, Lagrangian models are more efficient. A problem that has relatively few vehicles executing complicated maneuvers, if modeled in an Eulerian fashion, may require fine discretization of the space, resulting in many Eulerian cells that need to be modeled. In such circumstances, a model that focuses on the individual vehicles may be more efficient.

Optimization. Many models get built with the objective of optimizing a system. Some optimization problems are more naturally Eulerian, whereas others are more naturally Lagrangian. According to Bayen et al., when the objective is to come up with a more efficient use of airspace, rather than optimizing local trajectories of individual aircraft, we prefer an Eulerian model [47]. For our quarry example, we may be interested in minimizing the number of deployed haulers at a particular quarry site, and an Eulerian model such as that in Figure 3 can be used to evaluate scenarios. On the other hand, if the objective is to derive vehicle controllers that minimize the time that electric haulers spend at a recharging station, a Lagrangian model may be more appropriate. Jafari et al. present an Eulerian model in [2] to examine different configurations for the purpose of optimization in fleet management. In the context of the same project, the authors are working on a Lagrangian model of the quarry for optimization of the properties related to each machine.

Safety. Mobility is a safety-critical problem. Models can help ensure safety at multiple levels. A safety requirement for any such system is collision avoidance, and typically this requirement is addressed at multiple levels with redundant systems. At the lowest level, on-board sensors can be used with feedback control to ensure safe distances between vehicles. Figure 5 illustrates such low-level in a Lagrangian model. At a higher level, a track (or a sector) in an Eulerian model can be modeled as a critical section. The number of moving objects in each track can be controlled by the track itself or by a (semi-)centralized controller or coordinator. In Figure 3, for example, each track has a maximum capacity for vehicles determined by parameters shown in the figure. The safety of each track, in the sense of not being overloaded, can be evaluated using such a model. Tracks can be further subdivided in an Eulerian model so that each Eulerian node has a capacity of exactly one vehicle, and control strategies that ensure that this capacity is never violated are demonstrably safe. This method is similar to guaranteeing mutual exclusion of critical sections in distributed software systems using semaphores.

In air traffic control, the limits on the number of aircraft in a sector help reduce the risk of collisions. How to maintain these limits can be evaluated

using an Eulerian model [47]. At a much lower level, ACAS systems (Airborne Collision Avoidance Systems) such as TCAS demand Lagrangian models.

Another safety problem is ensuring that vehicles reach their destinations without running out of fuel or other critical resources. Eulerian models may be adequate if there is centralized control, as there might be in a quarry or a warehouse, for example. In the domains where free movement is possible in many directions in a large space, an Eulerian approach may not not be practical.

In Lagrangian models, for guaranteeing collision avoidance, the focus is on each moving object. Using multiple sensing and actuating devices, the moving object is aware of its surroundings and keeps a safe distance from the objects around. A mixed approach can be used where each track has a maximum capacity; using an Eulerian model we avoid overloading it, and each moving object is responsible for collision avoidance inside the track.

A good example is the use of unmanned aerial vehicles for civilian operations, which is rapidly increasing. We do not yet have solid approaches to guarantee safety of such large multi-agent systems if we allow unrestricted movements, as we might with a Lagrangian model. In [49], Chen et al. describe platoons of unmanned aerial vehicles flying on air highways. They argue for track-based structuring of the airspace and Eulerian models that allow for tractable analysis.

In general, there may be many different levels of abstractions in the airspace. For larger regions such as cities, air highways may prove beneficial (Eulerian model), and for a small region such as a neighborhood, perhaps unstructured flight is sufficiently safe (Lagrangian model). Further research is needed to better understand the parameters, such as the density of vehicles above which unstructured flight is no longer manageable, or other details like platoon size.

Control and infrastructure. For many mobility applications, modeling can help explore the trade offs between centralized and distributed control. One challenge is that the structure of models may be significantly different for the two cases, thereby making comparison more difficult. An Eulerian model often matches better with a centralized control strategy, particularly when the central controller maintains an overall view of the flow that individual vehicles are unable to form on their own. On the other hand, Lagrangian models may match better when control is decentralized.

Often, a combination of models is used. For example, the air traffic control (ATC) system works as a supervisory centralized controller with an Eulerian view of the airspace. On the other hand, a pilot's view is Lagrangian. In a city, the traffic light system is distinctly Eulerian, while self-driving vehicles are distinctly Lagrangian.

The moving objects in a mobility application have varying degrees of autonomy. Goods on a shelf in a warehouse and packets in a network have no autonomy, while self-driving cars have a great deal of autonomy. When dealing with more autonomy, a Lagrangian model is often better, but again it may complement an Eulerian model.

Effective Eulerian control often requires infrastructure investment. Traffic lights in cities are a good example. Sometime in the future, when all vehicles

are autonomous, traffic lights may become unnecessary. In closed (and not very big) environments, like warehouses, it is easier to build such an infrastructure. In such environments, sensors continually update the state of each track, and a centralized controller can dispatch waypoint commands to mobile agents. Once centralized knowledge of a configuration is formed, it can be made accessible to all the moving objects (or subsets of those), which can then use this additional information to supplement their own on-board sensor data to perform more effective Lagrangian control.

In an interesting blend of Lagrangian and Eulerian, Bayen et al. estimate the state of an Eulerian model of automotive traffic flow (a velocity field) from Lagrangian sensors (on-board GPS) on a subset of the vehicles [50]. They argue that this is more effective and less expensive than Eulerian sensing, where for example loop detectors or traffic cameras are placed at fixed locations.

The complexity and analyzability of control strategies may also be affected by the Lagrangian-Eulerian choice. According to Sun et al. [48], in the ATC domain, Eulerian models result in simpler, linear control-theoretic structure enables the use of standard methodologies to analyze them.

Adaptation. Mobility applications are required to be adaptive because conditions in the field and in the moving objects are constantly changing. An Eulerian model is often more appropriate when adapting to disruptions in the physical space, such as storms in air traffic control or blockages in a quarry. In such situations, the adaptation usually needs rerouting and rescheduling that may affect the whole system. On the other hand, when adapting to changes in battery status or fuel supply, a Lagrangian approach is likely more useful because the adaptation mostly affects the object itself. If the battery is low, the object may want to change its status to use less energy. It may also cause a more nonlocal adaptation, like rerouting and going towards a charging station. This example illustrates that in many cases Lagrangian models are better accompanied by some kind of Eulerian models to provide a view of the surroundings and hence provide a more solid basis for choosing adaptation policies.

Andersson et al. suggest several modeling dimensions for self-adaptive software systems in [51]. Their organization addresses "whether the adaptation is performed [or enforced] by a single component or distributed amongst several components," and ranges from centralized to decentralized; their scope is about "whether adaptation is localized or involves the entire system," and ranges from local to global. The above examples show how an Eulerian model better supports a decentralized and nonlocal adaptation, while a Lagrangian model can deal with more local adaptations.

In [40], Bagheri et al. use their coordinated adaptive actor model (CoodAA) [52], for performance evaluation and prediction of behavior of self-adaptive track-based traffic control systems. CoodAA is an Eulerian model and provides a framework for runtime analysis. If a change happens, the future behaviors of the model are explored and possible property violations are predicted. Appropriate policies can then be selected for adapting to the change. The cause of change may be anything in the physical space (like a storm for the ATC example) and

the adaptation policies are rerouting and rescheduling. The adaptation decision is made in a centralized form, and causes nonlocal changes. The framework is implemented in Ptolemy II using the discrete event director, similarly to the example in Figure 3.

The Eulerian model in CoodAA is faithful to the system, and the mapping of entities in the system to the model is simple and does not need any complex function or process. So, the changes that may happen in the system at runtime can easily be reflected in the model. This feature is important in building a runtime analysis framework because efficiency is crucial in runtime analysis. Also, the coordinated actor model reflects the central control in systems like ATC and railroad management systems.

Bagheri et al. propose a compositional runtime analysis technique, called Magnifier [53], that is based on the Eulerian view of the system. The technique is to focus on the point of change, and try to stop the propagation of effects of the change. If the propagation is inevitable, then zoom out and try the same technique. Magnifier technique can be used for adaptation of track-based flow management systems that use an Eulerian model.

An adaptive framework is proposed by Khakpour et al. in [54]. The framework is called PobSAM (Policy-based Self-Adaptive Model) and is an integration of algebraic formalisms and actor-based Rebeca models. A hierarchical extension of PobSAM is proposed by Khakpour et al. in [44]. In [44], the case study is on transportation service in the airport. Autonomous transport vehicles (ATVs) are signed in a transport scheduler service that collects passenger orders and gives tickets (pickup/drop positions, times) to the ATVs. ATVs are modeled in a Lagrangian way, where the ATVs have to collaborate and negotiate in competition on tickets, roads and charging stations. Eulerian models are added to bring in nonlocal control and an awareness of the surroundings; they for example can help the ATVs to avoid a congested area while transferring passengers. The hierarchical structure offers a form of controlled autonomy and balances agent autonomy and system controllability, for example to prevent unsafe situations caused by a selfish acting ATV.

4.5 Eulerian and Lagrangian Join Forces

Eulerian and Lagrangian patterns are complementary and can be combined effectively. A nice illustration of this is given by Claudel and Bayen [55], who introduce what they call "mixed Lagrangian-Eulerian sensing" for automotive traffic flow estimation. As we mentioned above, Eulerian sensing of traffic flow is based on installed physical infrastructure such as loop detectors, cameras, and speed radar. These are anchored to a physical location, and therefore provide information about a segment of roadway. Claudel and Bayen point out that these can be complemented with Lagrangian sensors, which travel with vehicles. In particular, they use GPS-enabled smartphones and show that even with a small percentage of vehicles so equipped, significant improvements in estimation are possible. In a personal communication, Bayen has also suggested Lagrangian *control*, not just sensing, of traffic flow. The idea is that even a small percentage

of self-driving vehicles on the road can be controlled so that they affect traffic flow, complementing Eulerian control techniques such as traffic lights on freeway entrance ramps.

Consider how we might combine the Eulerian model of the automated quarry in Figure 3 with the Lagrangian model in Figure 5. The Eulerian model provides a macroscopic view of the overall operation that can be used to define high-level strategies. A centralized controller could distribute instructions to haulers which will then carry out the instructions using low-level Lagrangian control. The Eulerian model mitigates congestion and minimizes queuing, while the Lagrangian model avoids collisions and monitors battery usage.

Anytime a (semi)autonomous agent needs to operate in a larger context, there is potential benefit from combining Eulerian and Lagrangian perspectives. The work of Khakpour et al. in [44] is a nice demonstration of this combination. Sensors fixed to infrastructure necessarily provide different information than sensors fixed to mobile agents, and controllers fixed to infrastructure necessarily provide a different kind of control than those fixed to mobile agents. Models need to reflect these complementary properties, and engineers need to consciously choose their modeling strategies cognizant of their strengths and weaknesses.

In a different context, for setting different parameters in a planning problem, an Eulerian model is used as a lightweight model prior to using a more detailed Lagrangian model. Castagnari et al. built an agent-based simulation framework for assessing the evolution of urban traffic after the introduction of new mobility services [46]. Each commuter is an agent in the model. The agent-based simulations are computationally expensive. So, they proposed their Eulerian model [35] which they used to estimate the simulation parameters for the Lagrangian model, and save expensive iterations of executing the Lagrangian model. They implemented a tool to map the inputs to the Lagrangian model to the inputs of the much lighter Eulerian model, and compared the outcomes to show the correlation.

Acknowledgements

We would like to thank Walid Taha for listening carefully to our discussion on track-as-actor and moving-object-as-actor design patterns and pointing out the relationship to modeling patterns that have been called Eulerian and Lagrangian, terms derived from fluid mechanics. We also would like to thank Claire Tomlin and Alex Bayen for the fruitful discussions and for pointing out relevant literature.

The work of the first author was supported in part by the US National Science Foundation (NSF), award #1446619 (Mathematical Theory of CPS), and the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by Avast, Denso, Ford, Siemens, and Toyota. The work of the second author was supported in part by DPAC Project (Dependable Platforms for Autonomous Systems and Control) at Malardalen University, Sweden, MACMa Project (Modeling and Analyzing Event-based Autonomous Systems) at Software Center,

Sweden, and the project Self-Adaptive Actors: SEADA (nr 163205-051) of the Icelandic Research Fund.

References

1. Volvo CE: Innovation at Volvo construction equipment Available at https://www.volvoce.com/global/en/this-is-volvo-ce/what-we-believe-in/innovation/, Retrieved August 24, 2018.
2. Jafari, A., Nair, J.J.S., Baumgart, S., Sirjani, M.: Safe and efficient fleet operation for autonomous machines: an actor-based approach. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018. (2018) 423–426
3. Manna, Z., Colón, M.A., Finkbeiner, B., Sipma, H.B., Uribe, T.E.: Abstraction and modular verification of infinite-state reactive systems. Requirements Targeting Software and Systems Engineering, Springer Berlin Heidelberg (1998) 273–292
4. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs, NJ, USA (1989)
5. van Glabbeek, R.J.: The linear time - branching time spectrum II. In: CONCUR '93, 4th International Conference on Concurrency Theory, Proceedings. (1993) 66–81
6. Lee, E.A.: Plato and the Nerd — The Creative Partnership of Humans and Technology. MIT Press (2017)
7. Sirjani, M.: Power is overrated, go for friendliness! expressiveness, faithfulness and usability in modeling - the actor experience. In: Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. Lecture Notes in Computer Science 10760 (2018) 424–449
8. Box, G.E.P., Draper, N.R.: Empirical Model-Building and Response Surfaces. Wiley Series in Probability and Statistics. Wiley (1987)
9. Boehm, B.W.: Verifying and validating software requirements and design specifications. IEEE Software (1984) 75–88
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages (POPL), ACM Press (1977) 238–252
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
12. Brooks, R.A.: Artificial life and real robots. In Varela, F.J., Bourgine, P., eds.: Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, Cambridge, MA, MIT Press (1992) 3–10
13. Lee, E.A.: Fundamental limits of cyber-physical systems modeling. ACM Transactions on Cyber-Physical Systems **1**(1) (2016) 26
14. Hewitt, C.: Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence **8**(3) (1977) 323–363
15. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7**(1) (1997) 1–72
16. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inform. **63**(4) (2004) 385–410
17. Sirjani, M.: Rebeca: Theory, applications, and tools. In: Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006. (2006) 102–126
18. Sharifi, Z., Mosaffa, M., Mohammadi, S., Sirjani, M.: Functional and performance analysis of network-on-chips using actor-based modeling and formal verification. ECEASST **66** (2013)

19. Khamespanah, E., Mechitov, K., Sirjani, M., Agha, G.A.: Schedulability analysis of distributed real-time sensor network applications using actor-based model checking. In: Model Checking Software - 23rd International Symposium, SPIN 2016, Proceedings. (2016) 165–181
20. Jahandoust, G., Ghassemi, F.: An adaptive sinkhole aware algorithm in wireless sensor networks. Ad Hoc Networks **59** (2017) 24–34
21. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP Congress 74, North-Holland Publishing Co. (1974) 471–475
22. Dennis, J.B.: First version data flow procedure language. Report MAC TM61, MIT Laboratory for Computer Science (1974)
23. Benveniste, A., Caspi, P., Le Guernic, P., Halbwachs, N.: Data-flow synchronous languages. In Bakker, J.W.d., Roever, W.P.d., Rozenberg, G., eds.: A Decade of Concurrency — Reflections and Perspectives. Volume 803 of LNCS., Berlin, Springer-Verlag (1994) 1–45
24. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In Gilchrist, B., ed.: Information Processing, North-Holland Publishing Co. (1977) 993–998
25. Lee, E.A., Matsikoudis, E.: The semantics of dataflow with firing. In Huet, G., Plotkin, G., Lévy, J.J., Bertot, Y., eds.: From Semantics to Computer Science: Essays in memory of Gilles Kahn, Cambridge University Press (2009)
26. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE **91**(2) (2003) 127–144
27. Ptolemaeus, C.: System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley, CA (2014)
28. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation. 2nd edn. Academic Press (2000)
29. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14**(3) (2004) 329–366
30. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8) (1978) 666–677
31. Tiller, M.M.: Introduction to Physical Modeling with Modelica. Kluwer Academic Publishers (2001)
32. Tripakis, S., Stergiou, C., Shaver, C., Lee, E.A.: A modular formal semantics for ptolemy. Mathematical Structures in Computer Science **23**(Special Issue 04) (2013) 834–881
33. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., IV, C.T., Nordstrom, G., Sprinkle, J., Volgyesi., P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing. (2001)
34. Batchelor, G.K.: An introduction to fluid dynamics. Cambridge University Press, Cambridge, UK (1973)
35. Castagnari, C., de Berardinis, J., Forcina, G., Jafari, A., Sirjani, M.: Lightweight preprocessing for agent-based simulation of smart mobility initiatives. In: Software Engineering and Formal Methods - SEFM 2017 Collocated Workshop: FOCLASA, Revised Selected Papers. (2017) 541–557
36. Lee, E.A., Liu, J., Muliadi, L., Zheng, H.: Discrete-event models. In Ptolemaeus, C., ed.: System Design, Modeling, and Simulation using Ptolemy II, Berkeley, CA, Ptolemy.org (2014)
37. Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci. Comput. Program. **98** (2015) 184–204

38. Sirjani, M., Khamespanah, E.: On time actors. In: Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday. (2016) 373–392
39. Reed, G.M., Roscoe, A.W.: A timed model for communicating sequential processes. Theoretical Computer Science **58**(1-3) (1988) 249–261
40. Bagheri, M., Sirjani, M., Khamespanah, E., Khakpour, N., Akkaya, I., Movaghar, A., Lee, E.A.: Coordinated actor model of self-adaptive track-based traffic control systems. Journal of Systems and Software **143** (2018) 116–139
41. Cardoso, J., Lee, E.A., Liu, J., Zheng, H.: Continuous-time models. In Ptolemaeus, C., ed.: System Design, Modeling, and Simulation using Ptolemy II, Berkeley, CA, Ptolemy.org (2014)
42. Menon, P.K., Sweriduk, G.D., Bilimoria, K.: A new approach for modeling, analysis and control of air traffic flow. In: AIAA Conference on Guidance, Navigation, and Control. (2004)
43. Baldwin, P., Kohli, S., Lee, E.A., Liu, X., Zhao, Y.: Modeling of sensor nets in Ptolemy II. In: Information Processing in Sensor Networks (IPSN). (2004)
44. Khakpour, N., Jalili, S., Sirjani, M., Goltz, U., Abolhasanzadeh, B.: HPobSAM for modeling and analyzing IT ecosystems - through a case study. Journal of Systems and Software **85**(12) (2012) 2770–2784
45. Jayanthi Surendran Nair, J.: Modelling and analysing collaborating heavy machines. Master's thesis, Mälardalen University, School of Innovation, Design and Engineering (2017)
46. Castagnari, C., Corradini, F., Angelis, F.D., de Berardinis, J., Forcina, G., Polini, A.: Tangramob: an agent-based simulation framework for validating urban smart mobility solutions. CoRR **abs/1805.10906** (2018)
47. Bayen, A.M., Raffard, R.L., Tomlin, C.J.: Eulerian network model of air traffic flow in congested areas. In: American Control Conference. (2004)
48. Sun, D., Yang, S., Strub, I.S., Bayen, A.M., Sridhar, B., Sheth, K.: Eulerian trilogy. In: American Institute of Aeronautics and Astronautics. (2006)
49. Chen, M., Hu, Q., Fisac, J.F., Akametalu, K., Mackin, C., Tomlin, C.J.: Reachability-based safety and goal satisfaction of unmanned aerial platoons on air highways. Journal of Guidance, Control, and Dynamics **40**(6) (2017) 1360–1373
50. Bayen, A., Raffard, R., Tomlin, C.: Lagrangian sensing: traffic estimation with mobile devices. American Control Conference (2009)
51. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]. (2009) 27–47
52. Bagheri, M., Akkaya, I., Khamespanah, E., Khakpour, N., Sirjani, M., Movaghar, A., Lee, E.A.: Coordinated actors for reliable self-adaptive systems. In: Formal Aspects of Component Software - 13th International Conference, FACS 2016. (2016) 241–259
53. Bagheri, M., Khamespanah, E., Sirjani, M., Movaghar, A., Lee, E.A.: Runtime compositional analysis of track-based traffic control systems. SIGBED Review **14**(3) (2017) 38–39
54. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: PobSAM: Policy-based managing of actors in self-adaptive systems. Electr. Notes Theor. Comput. Sci. **263** (2010) 129–143
55. Claudel, C.G., Bayen, A.: Guaranteed bounds for traffic flow parameters estimation using mixed Lagrangian-Eulerian sensing. 2008 46th Annual Allerton Conference on Communication, Control, and Computing (2008) 636–645