Environmental Modelling and Software 122 (2019) 104553

Contents lists available at ScienceDirect

Environmental Modelling and Software

journal homepage: http://www.elsevier.com/locate/envsoft

A physical data model for spatio-temporal objects

Kor de Jong [a,b,*], Derek Karssenberg [a]

[a] Department of Physical Geography, Faculty of Geosciences, Utrecht University, Princetonlaan 8A, Utrecht, 3584 CB, The Netherlands
[b] Department of Information and Computing Sciences, Faculty of Science, Utrecht University, Princetonplein 5, Utrecht, 3584 CC, The Netherlands

ARTICLE INFO

Keywords:
Simulation
Modelling
Field
Object
HDF5
LUE

ABSTRACT

Modellers simulating the state of the physical and biological environment using agents and fields, face the challenge of storing the state variables, like the distribution of biomass and the location and properties of animals. These variables differ from each other, depending on how exactly they represent their state. In this paper we describe an approach for storing multiple kinds of representations of model state variables using a unified physical data model. We identified a set of aspects in which these representations differ from each other and which have an implication for the data model. Based on these aspects we defined a physical data model for storing spatio-temporal objects, which we implemented in an open source software library. We illustrate how the resulting data model can be used to store multiple kinds of model state variables and explain how this data model can be useful in environmental modelling software.

1. Introduction

When simulating the state of the physical and biological environment using a computer model, model developers face the challenge of storing this state, as represented by the model variables, in one or more datasets for post-processing. Depending on how the model variables represent state, storing each of them may require different physical data models (dataset formats or database designs). For example, when modelling the influence of grazing deer on the spatial distribution of biomass, the state to be stored might be the continuous distribution of the biomass through time and space, the changing location in space of each deer, the changing non-spatial weight of each deer, and the sex of each deer, which is not dependent on time and space. The modeller might in this case decide to store biomass using a collection of rasters (one raster per simulated location in time), stored in one of the raster formats supported by the Geospatial Data Abstraction Library (GDAL, GDAL/OGR contributors (2019)), animal location using a collection of spatial points per animal stored in one of the vector formats supported by GDAL, animal weight by a collection of floating point numbers per animal stored in an SQLite database (SQLite developers (2000–2019)), and sex by a single code per animal also stored in the SQLite database. Since GDAL does not support time, some ad hoc convention is needed in this case to relate information in the datasets to simulated time. An additional convention is needed to relate the information about the simulated deer in the various datasets to each other. Having to use multiple physical data models and conventions is inconvenient for modellers, error-prone and results in models that are less easy to maintain.

Which kinds of representations of state variables are used in environmental models depends on various factors, of which the modelling paradigm used is an important one. Two main paradigms for modelling environmental processes are field-based modelling and agent-based modelling (Filatova et al. (2013); Goodchild (2013); Parker (2005)). In field-based modelling, modelled systems are represented by continuous spatial fields of information. These fields are often represented by rasters, like a raster containing amounts of biomass in an area. In agent-based modelling, systems are represented by interacting discrete objects of information that are mostly bounded in time and space. These entities can be representations of physical real-world phenomena, like deer, but they can also represent organizational entities, like stakeholders or companies.

Many field-based models exist in which agents are manipulated and vice-versa, such as in the example of the deer – biomass system. Instead of choosing between a field-based or agent-based modelling approach, often an integrated approach is taken, in which both continuous fields and discrete agents are manipulated, examples of which can be found in Bennett and Tang (2006), Castilla-Rho et al. (2015), Sbihi et al. (2015), Schelhaas et al. (2007) and Schippers et al. (2014).

In both field- and agent-based models, and especially in integrated field- and agent-based models, multiple kinds of representations of state

* Corresponding author. Department of Physical Geography, Faculty of Geosciences, Utrecht University, Princetonlaan 8A, Utrecht, 3584 CB, The Netherlands.
E-mail addresses: k.dejong1@uu.nl (K. de Jong), d.karssenberg@uu.nl (D. Karssenberg).

https://doi.org/10.1016/j.envsoft.2019.104553
Received 16 July 2019; Received in revised form 4 October 2019; Accepted 9 October 2019
Available online 16 October 2019
1364-8152/© 2019 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

variables are manipulated. Various physical data models exist for storing these kinds of state variables. APIs for storing rasters and objects are, for instance, provided by GDAL (GDAL/OGR contributors (2019)), HDF5 (The HDF Group (1997–2019)), netCDF4 (UNIDATA (2008–2018)), rasdaman (Baumann et al. (1998)), SciDB (Becla et al. (2013)) and PostGIS (PostGIS development community (2019)). There are important shortcomings of current options for storing state variables:

- Most physical data models support either the storage of rasters or of objects, but not both. When creating integrated models, the modeller needs to be familiar with multiple data models, store information in multiple datasets, and may have to store information like the locations in time multiple times. This results in a complex, error-prone, and inefficient workflow.
- Some physical data models do not support random access. This implies that when a certain piece of information needs to be found, possibly the whole dataset needs to be read first. This increases the runtime of models and their memory requirements.
- Existing physical data models often have limited support for storing locations in time, or only support locations in time relative to the Gregorian calendar. Forward models working with very small or long time scales, ranging from nano-seconds to millions of years, then need to resort to ad hoc conventions for storing locations in time.
- Some physical data models do not benefit from the increased amount of resources offered by high performance computing (HPC) facilities. For example, they do not have support for parallel I/O. Also, for performance reasons, some HPC facilities do not support the installation of server processes. This precludes the use of data models that require a database management system (e.g. PostGIS, rasdaman, SciDB).
- Some physical data models have limitations with respect to the number of objects that can be stored, or the size of the objects that can be stored. Ideally, the number of objects and the size of the objects should only be limited by limitations of the hardware, not by limitations of the data model.

In this work we tried to answer the question whether it is possible to design a unified physical data model that can be used to store different simulation model state variable representations in an integrated way. Such a data model should ideally simplify the way environmental modellers handle the storage of model state, and not suffer from the above mentioned shortcomings of current data models. Also, the data model should be receptive of features currently not found in some of the popular data models, like the handling of simulated time, mobility and collections of objects.

How model state is organized in a dataset might influence the way state is represented by model variables at runtime. Whenever possible, time consuming conversions must be prevented, for example. Although in this work we focus on the physical data model and not on the representation of model state variables, our results align with the conceptual data model described in de Bakker et al. (2017), which could be used as a basis for representing model state variables.

In order to answer our research question, we first looked at the different kinds of information that are relevant to store in a unified data model. This is described in section 2. The objective was to extract the commonalities from different kinds of representations of state variables and use these as a basis for our data model. This work resulted in a definition of spatio-temporal objects, which can be found in the same section. This definition of spatio-temporal objects allowed us to frame different kinds of model entities as variations of a single kind of spatio-temporal object.

Two important factors guided our data model design: performance and the management of complexity of the data model. In the case of simulation models that read and write much data, the total runtime can become a performance bottleneck. For a data model to be relevant in environmental modelling, it must offer good performance. For this we applied the principle of locality to our data model. This is described in section 3.1.

When permuting the different approaches for representing locations in time, locations in space, and properties, we initially ended up with a large number of different kinds of representations. We concluded that it was unfeasible to design a physical data model for each of these. Instead, we focused on identifying a lowest abstraction level for representing abstract temporal object information and built our data model on top of that, using abstraction levels of increasing functionality (Fig. 1). Arrays of temporal information form the first and lowest layer of a stack of three layers of abstractions. This layer is described in section 3.1. In the second layer, described in section 3.2, various kinds of spatio-temporal object information are defined, in terms of the arrays of temporal information from the first layer. In the third and highest layer of abstraction, the spatio-temporal objects are defined, in terms of the abstractions in the second layer. This layer is described in section 3.3. In section 4 we describe the implementation of our data model. Using this implementation, we were able, in a case study, to see how well different kinds of state variables from environmental models could be represented. Section 5 shows how this was done.

This work resulted in a physical data model for storing spatio-temporal objects, on top of the HDF5 data model (Fig. 1). The data model is implemented in C++ and is currently exposed through a C++ API and a Python API. Section 8 contains information about how to obtain a copy of the source code.

2. Spatio-temporal objects

A unified physical data model for storing state variables from environmental models must be able to represent the different kinds of representations of state variables in a uniform manner. With this we mean that it must be possible to view each kind of state variable as an example of a more general kind of state variable. For example, it must be possible to view a global variable representing growth rate for grass, a collection of multiple wandering deer (Fig. 2a) and a single temporal biomass field (Fig. 2b) as similar in terms of the physical data model.

In order to come up with an approach for such a unified physical data model, we first looked at the differences and commonalities between the different kinds of information represented by these state variables. In the simplest case, a state variable represents a single value that does not change during the simulation and is not located at a specific location in space. A floating point value representing the Earth's standard gravity in a plot-scale environmental model is an example of this. At the other extreme, a state variable might represent a collection of objects, each of which is located at a specific location in space and has a property value, both of which change during the simulation. A variable containing information about the locations in space and weights of a collection of wandering deer is an example of this. Besides these two extreme cases there are many more kinds of state variables, differing with respect to the size and variability of the collection of objects they represent, whether or not information changes through time or space, and the kind of property values they contain. We will now describe each of these aspects in turn, in order to end up with an approach to viewing each different kind of state variable as an example of a more general one.

We define an object as a uniquely identifiable entity whose state is simulated by a model. Objects of the same kind (for example, deer) are grouped in collections. Objects can be added and removed from such a collection (for example, deer are born and die). So, during a simulation, the set of objects that participate in the calculations may change. We call this changing set of participating objects the set of active objects, or the active set. The active set represents *who* are active. Although we use concrete examples for objects in this discussion, we want to stress that we assume as little as possible about the nature of the objects; what a collection of objects represents is up to the application (that is: the simulation model).

Current kinds of state variables do not always represent collections of

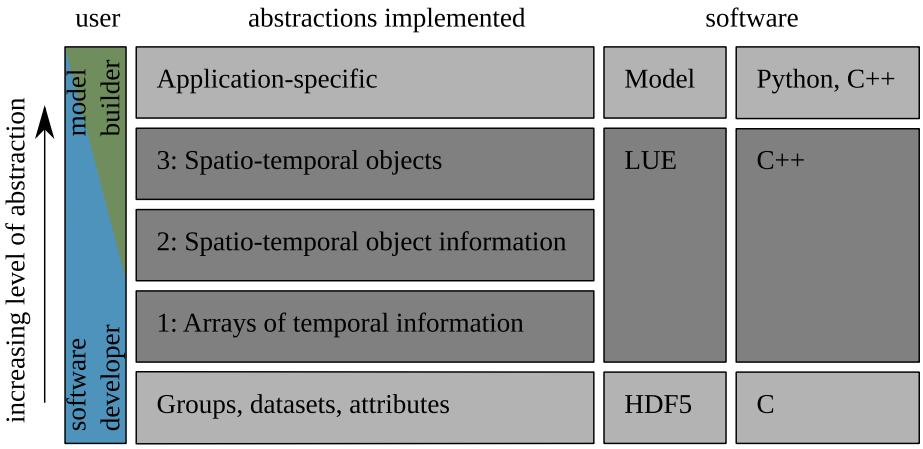
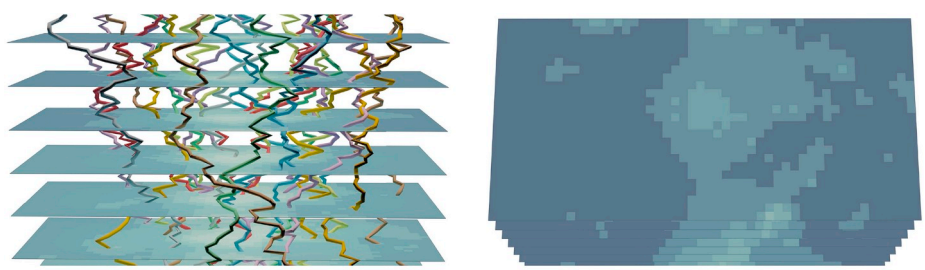


Fig. 1. The concepts implemented in different abstraction layers of the software stack implementing the physical data model. The right columns show the names of the corresponding software implementing the abstraction layers and the programming languages used to implement them. The left column shows the expected user of the concepts in each abstraction layer. The dark grey colours represent the three layers of the software stack discussed in this paper. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)



(a) (b)

Fig. 2. Example output of an integrated simulation model with objects (deer) and a field (environment). The third dimension, pointing up, is used here to represent time. Fields are shown for one in ten time steps. Darker coloured cells contain more biomass. a: Locations of deer vary through time. Deer wander from cells containing less biomass to those containing more. b: Biomass variation through time and space. Cells where deer have grazed recently contain less biomass.

objects, as in the case with the floating point variable representing the Earth's standard gravity mentioned above. As a first generalization step towards unifying all kinds of state variables, we decided that all information represented by them must be associated with one or more objects. In the case of the Earth's standard gravity, this object represents the planet Earth.

All our objects have a presence in time and in space. With presence in time we mean that for each object, information must be stored about *when* the object was active. Objects might be active at very specific locations in time, or during longer periods. For example, it is common for objects representing living things, like deer, to be active from simulated birth to death. With presence in space we mean that for each location in time that an object was active, information must be stored about *where* the object was active. Objects might be active at specific locations in space, or in regions with a spatial extent. A bird's location can be represented by a point, and a tree's crown by a polygon, for example. Additionally, objects can be stationary, in which case the location in space is fixed through time, or mobile.

We use the concept of properties to represent *what* is present at locations in time and space that an object is active. For example, a relevant property manipulated by the model might be the weight of each deer. All objects in a collection have the same set of properties. So, either weight is stored for each deer, or weight is not stored at all.

For a single object, there are multiple ways to represent a property value, depending on the kind of property. A deer's weight can be represented by a single number, a natural park's biomass field by a 2*D* array, and a bird's direction and speed by a 1*D* Euclidean vector. The final step we took towards unifying state variables, is to represent all individual property values by *nD* arrays, where the dimensionality of these arrays is the same per property.

Given these generalization steps and in the context of our physical data model, we can now define spatio-temporal objects as uniquely identifiable objects for which locations in time, locations in space and properties are stored for those locations in time that the objects where active. Our goal is that all model state variable representations, ranging from scalar constants to mobile agents with discretized properties, can be framed as collections of such spatio-temporal objects. A physical data model capable of storing such collections of objects is then capable of storing such a diverse set of state variable representations. In the next section we describe our approach to representing these objects in a physical data model.

3. Storing spatio-temporal objects

3.1. Arrays of temporal information (abstraction level 1)

Arrays of temporal information are at the lowest level of abstraction of our data model (layer 1 in figure (1)). They represent any kind of information for which variation through time has to be stored in the data model. At this abstraction level it is not relevant what kind of information is stored exactly. This could be the ID of an object, a coordinate in time or space, or an object's property value.

The principle that guided our approach was the principle of locality (Patterson and Hennessy (2008)). In the context of modelling, this principle states that, at any moment in time, a model accesses a relatively small portion of the data. There are two types of locality. Temporal locality means that if a model uses data, it will probably use the same data again soon. Spatial locality means that if a model uses data, nearby data (in terms of memory addresses) will probably be used soon.

The principle of locality and the fact that different kinds of memory differ a lot in speed and price, with the fastest memory being the most expensive, led to the memory hierarchy found in computers. The fastest

memory is located near the CPU cores and the slowest memory is located further away, on SSD drives and spinning disks, for example. In between multiple levels of caches are available, like the caches in the storage devices themselves, the main memory and the CPU caches.

Memory is copied from lower (distant to the CPU cores) levels to upper levels in blocks of multiple values. Besides the values requested, nearby values are also copied, under the assumption that the CPU will probably need those values soon, too (spatial locality). For software to benefit from the memory caches in the memory hierarchy, it must store the values compactly, in the order in which they are accessed. That way, the caches are filled with relevant values more often. This approach to organize the values based on what is beneficial in the context of the principle of locality and the memory hierarchy is called data-oriented design (Sharp (1980); Fabian (2018)). The goal of data oriented design is to decrease the idle time of CPU cores, by increasing the likelihood that required values are in the nearby memory caches.

To understand what this means in the context of forward modelling, we looked at the data access patterns in forward models defined using three modelling environments: PCRaster field-based modelling environment (Karssenberg et al. (2010)), NetLogo agent-based modelling environment (Wilensky (1999)) and Mason agent-based modelling environment (Luke et al. (2005)). In these models we can typically (but not always) identify a number of nested iterations (from outer to inner):

1. *Time*: A model iterates through time until the final state of the modelled system has been calculated. Each system's state is calculated based on the system's previous state(s).
2. *Operations*: Each individual state is calculated by performing a number of operations in sequence. This set of operations is the core of the model and implements the modelled processes.
3. *Objects*: Each individual operation iterates over a set of objects to calculate some result. In this calculation, the operation typically uses a small number of object properties. An example of this is an operation that calculates the health of each deer in a collection, based on each deer's age and weight.

This iteration scheme assumes that a simulation model consists of a sequence of operations performed iteratively through time on (selections of) objects. This is a relatively simple approach, but it has some benefits we think are good to have, the most important one being that the approach naturally aligns with data-oriented design. The iteration scheme suggests a preferred ordering for the storage of information. Values for (only) the same kind of object information, like the values of a single property for all objects, should be stored close to each other (in terms of memory addresses). These collections of values should then be ordered by time. This will increase the likelihood that modelling software benefits from the caching in the memory hierarchy. When an operation iterates over a collection of objects, reading specific property values, the memory caches will likely be filled by relevant property values, not including information that is not currently needed.

Based on the principle of locality, data orientation and the iteration scheme often used in forward modelling, we decided to store information for each specific kind of object information as close together as possible and sorted by time. This means that, for example, for all objects all locations in space are stored close together, and property values of a single property are stored close together as well. This contrasts with an approach where, for each object individually, all information is stored together.

The most compact way to store multiple values in computer memory is as an array, whose element values are stored in contiguous memory locations. Therefore, we concentrated on finding ways to organize spatio-temporal object information as arrays. Each array has a shape, which is the collection of the size of each of the array's dimensions. The number of these dimensions is the array's rank. For example, a spatial raster can be represented by an array with rank two whose shape is equal to the number of rows and columns. A single array can contain multiple arrays with a smaller rank. For example, multiple spatial rasters associated with multiple locations in time can be represented by an array with rank three, whose shape is equal to the number of locations in time, rows and columns. This packing of arrays in larger arrays results in the most compact way to store object related information, which is beneficial in the light of the above mentioned memory caching.

Whether or not individual arrays containing information per object and per location in time can be packed in larger arrays depends on the shape of each individual array. Here we define object arrays as arrays containing a piece of information for a single object and for a single location in time. This could be an object's ID, or a property value, for example. We call the array resulting from packing one or more object arrays in larger arrays the value array.

Multiple object arrays can be tightly packed in value arrays when the shapes of the object arrays are equal. Object arrays containing object IDs are all 0*D* arrays that can be packed in a single 1*D* value array. Object arrays containing 2*D* arrays representing, for instance, the biomass property of multiple natural park areas cannot be packed in a single 3*D* value array.

The criteria for deciding whether or not object arrays can be packed in value arrays are shown in Table 1. Permuting these criteria resulted in six approaches for packing object arrays into value arrays. Here, it is not yet relevant what exactly is represented by the information in the object arrays (for example object IDs, locations in time, or property values). What is relevant is that there are different kinds of object arrays and that the differences between them determine how to represent them in a physical data model. Our hypothesis is that with a limited set of six kinds of relatively low-level data models for storing arrays of temporal information, we can represent the much larger set of different kinds of higher-level spatio-temporal objects.

Six figures corresponding with the six object array kinds from Table 1 illustrate the object array kinds and their packing into value arrays. Object arrays of different objects can have the same shape (Figs. 3, 5 and 7) or a different shape (Figs. 4, 6 and 8). Note that the object arrays shown in the figures are small 1*D* arrays, but in reality object arrays can be very large and have a very different shape. Object arrays for temporal information can have a constant shape (Figs. 5 and 6) or a variable shape (Figs. 7 and 8).

Packing object arrays into value arrays is not possible for every kind of object array. Fig. 8 shows that for each object array and each location in time, the object information needs to be stored in a separate value array. This potentially results in a large number of value arrays. The approach taken in this extreme case will not be beneficial for the memory caching. On the other hand, Figs. 3 and 5 show that all object arrays (of all locations in time in the case of Fig. 5) can be stored in a single value array (ordered by location in time). This approach will potentially be beneficial for the memory caching.

The collection of active objects often changes during a simulation (section 2). In case of the packing of the temporal object array kinds

Table 1
Taxonomy of object arrays. Permuting object array value variability through time (*constant value* versus *variable value*), object array shape difference per object (*same shape array* versus *different shape array*) and object array shape variability through time (*constant shape array* versus *variable shape array*) results in six kinds of object arrays. The terms in the last column are used in the text.

Do the object arrays contain information that changes through time?	Does the shape of the object arrays differ per object?	Does the shape of the object arrays change through time?	
no	no	no	*constant value* × *same shape array*
no	yes	no	*constant value* × *different shape array*
yes	no	no	*variable value* × *same constant shape array*
yes	no	yes	*variable value* × *same variable shape array*
yes	yes	no	*variable value* × *different constant shape array*
yes	yes	yes	*variable value* × *different variable shape array*

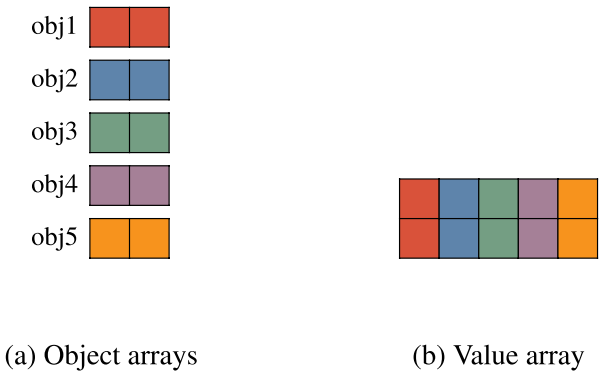


Fig. 3. *Constant value* × *same shape array*: Object arrays for different objects have the same shape. Only a single object array needs to be stored per object. All object arrays are packed into a single value array. In this example, five 1*D* object arrays for five different objects are shown. Each object array contains two values, for example, an x and y-coordinate pair. The value array is a 2*D* array of shape $(5, 2)$. Applicability: object IDs, stationary space points and constant *nD* property values where each array has the same shape, like scalars and Euclidean vectors.

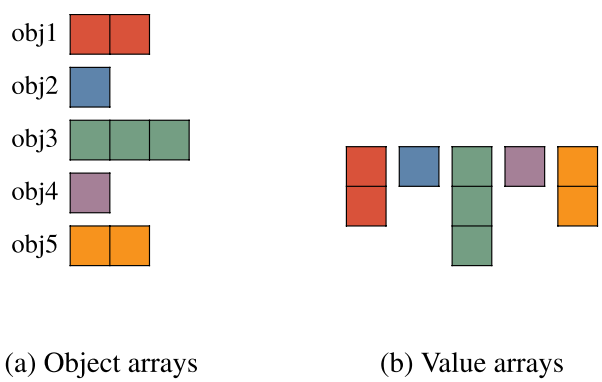


Fig. 4. *Constant value* × *different shape array*: Object arrays for different objects have different shapes. Only a single object array needs to be stored per object. Each object array is packed into a separate value array. Applicability: constant *nD* property values where each array has a different shape, like spatial rasters for differently discretized areas.

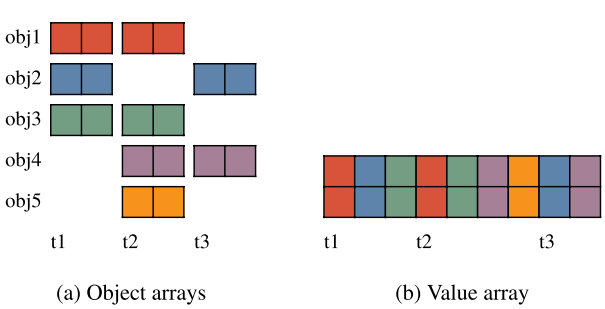


Fig. 5. *Variable value* × *same constant shape array*: Object arrays for different objects have the same shape, which does not change through time. All object arrays are packed into a single value array. Object arrays with the same colour are related to the same object. Applicability: mobile space points, temporal *nD* property values, like scalars and Euclidean vectors. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

shown in Figs. 5–8, this information is not part of the value arrays; it has to be explicitly stored somewhere else in the data model. For details about our approach for doing this for each of the temporal object array kinds, we refer to section AppendixA.

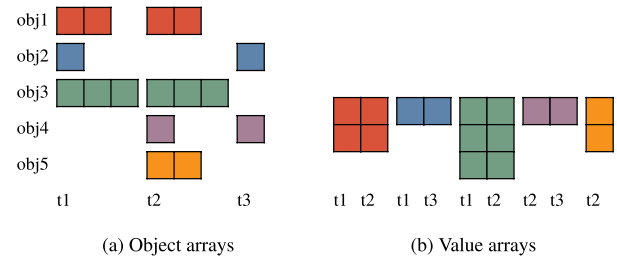


Fig. 6. *Variable value* × *different constant shape array*: Object arrays for different objects have different shapes, which does not change through time. Per object, object arrays for multiple locations in time are packed into a single value array. Applicability: temporal *nD* property values, like spatial rasters for differently discretized areas.

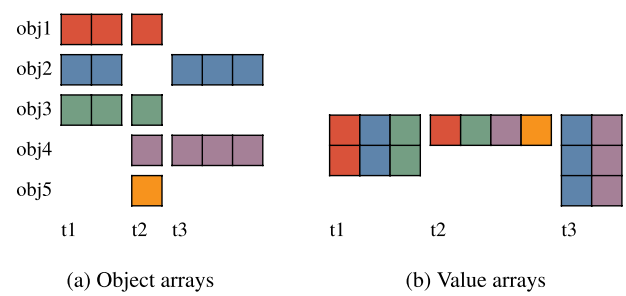


Fig. 7. *Variable value* × *same variable shape array*: Object arrays for different objects have the same shape, which changes through time. Per location in time, object arrays for multiple objects are packed into a single value array. Example: temporal *nD* property values, like spatial rasters for equally discretized areas and for which this discretization changes through time.

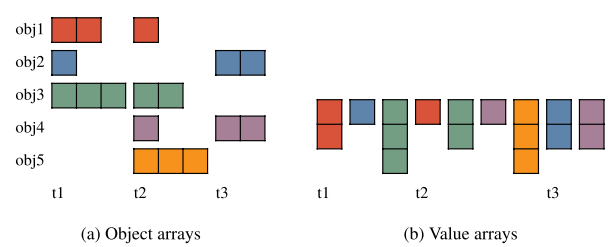


Fig. 8. *Variable value* × *different variable shape array*: Object arrays for different objects have different shapes, which changes through time. Each object array is packed into a separate value array. Applicability: temporal *nD* property values, like spatial rasters for differently discretized areas and for which this discretization changes through time.

3.2. Spatio-temporal object information (abstraction level 2)

In section 2 we defined spatio-temporal objects as uniquely identifiable objects for which locations in time, locations in space and properties are stored for those locations in time that the objects where active. We will now describe for each of these pieces of object-information how we decided to represent it in terms of the six kinds of object arrays described in the previous section. These abstractions correspond with the second abstraction layer implemented in our data model (Fig. 1).

3.2.1. Identity

Object identity can be represented by a unique unsigned integer value which, represented as an array, corresponds with a 0*D* array with an empty shape. This shape is the same for all objects (*same shape array*). In case of object information that does not change through time, the associated object identity can be represented by the *constant value* × *same shape array* object array kind (Fig. 9a). In case of object information that does change through time, for each location in time the IDs of the

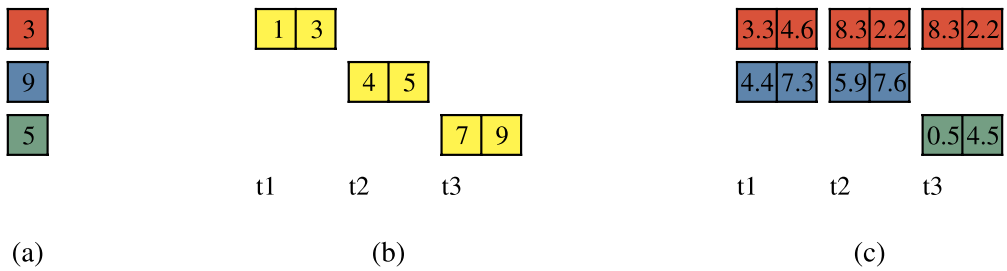


Fig. 9. Examples of object arrays for representing spatio-temporal object information. a: IDs associated with constant object information of three objects (see Fig. 3 for packing). b: Time boxes to be associated with collections of objects that are active (see Fig. 5 for packing). c: Mobile space points of three objects (see Fig. 5 for packing).

active objects can be represented by the *variable value* × *same constant shape array* object array kind (Fig. 5, sections AppendixA.1, AppendixA.2 and AppendixA.3), or by encoding the IDs of the active objects in the meta-information of the value array used to store each object array (section AppendixA.4).

3.2.2. Locations in time

We implemented three different approaches for storing locations in time that objects can be active: time points, time boxes and time cells (Fig. 10). Which objects are actually active at these locations is handled by object tracking (section AppendixA). A time point is used to represent a specific location in time objects can be active. A time box is used to represent a period of time objects can be active. It is defined by a start time point and an end time point. Time cells are used to discretize time boxes for more fine grained tracking of object activity.

We represent time points by positive integral numbers representing an amount of time (duration) since an epoch. Details about this approach for handling time can be found in AppendixB. Time durations can be represented by 1*D* object arrays with shape (1), and this shape does not change through time. This matches the *variable value* × *same constant shape array* object array kind (Fig. 5).

Time boxes can be handled similarly as time points, but instead of storing a single time point per location in time, two increasing time points need to be stored. So, a single time box can be represented by a 1*D* array with two durations in it, representing the start and end time points. The shape of this array is (2) and this shape stays the same through time (Fig. 9b). This matches the *variable value* × *same constant shape array* object array kind (Fig. 5).

For time cells, we store an additional count for each time box. Counts are unsigned integers, represented by *variable value* × *same constant shape array* object arrays.

3.2.3. Locations in space

Representing locations in space where an object is active can be done in multiple ways. For example, the Simple Feature Access standard (Open Geospatial Consortium (2011)), defines points, lines, triangles, and multi-polygons, amongst others. We implemented a sub-set of these approaches: space points and space boxes, which we extended to support defining locations in 1*D*, 2*D* and 3*D* space. Both stationary and mobile locations in space are supported.

A space point is used to represent a specific location in space where an object is active. A space box is used to represent a linear, rectangular or cuboidal region of space where an object is active. It is defined by two diagonally opposite space points.

A space point can be represented by a coordinate in each spatial dimension. A coordinate can be represented by a number (integer or floating point). The corresponding object array of a space point, then, is a 1*D* array with a shape equal to the rank of the space (1, 2, or 3 dimensions). For each point, this array contains the coordinates. Stationary space points are represented by the *constant value* × *same shape array* object array kind (Fig. 3). Since object arrays containing space points for multiple locations in time have the same shape (the same 1*D* array as in the stationary case), mobile space points are represented by the *variable value* × *same constant shape array* object array kind (Fig. 9c).

As with space points, a space box can be stored in a 1*D* object array, but this time the shape is equal to twice the shape of a single space point. The object array kinds for representing this information are the same as those for representing space points: *constant value* × *same shape array* object array kind (Fig. 3) in case of stationary space boxes, and *variable value* × *same constant shape array* object array kind (Fig. 5) in case of mobile space boxes.

3.2.4. Properties

In environmental modelling various kinds of property values are used to represent an object's trait. These kinds of values differ from each other based on whether or not these values vary through time, whether or not the shape of the arrays representing the values differ per object, and whether or not the shape of these arrays varies through time. For example, the weight property of each deer can be represented by a single number, which varies through time, while the surface elevation property of a natural park area can be represented by a constant or variable 2*D* numeric array, depending on whether the elevation changes through time.

The criteria with which property values can be classified match the ones we identified for our six kinds of object arrays, described in section 3.1. In the previous sections, specific kinds of object arrays were used to represent specific kinds of object information. In the case of properties, all six object array kinds from our taxonomy can be used for storing property values. This way our data model can support a wide variety of kinds of property values used in environmental modelling.

As an example, the weight of deer through time can be stored in a *variable value* × *same constant shape array* object array kind (Fig. 5). For each location in time that each deer was active, the value array will contain a floating point number representing the weight. As another example, a simulation model modelling the evolution of the elevation of the land surface of multiple research area objects can store 2*D* digital elevation models as *variable value* × *different constant shape array* object arrays (Fig. 6, assuming the research areas are stationary and have a different, but constant shape).

Property values can be discretized through either or both time and space. We decided to store information about how property values are discretized as a property itself. In case of a spatial raster, for example, information about the number of rows and columns a 2*D* property value is discretized in is stored in a separate property, and linked to the property being discretized. The advantage of handling information about a discretization as a property is that this information itself can vary through time and potentially even through space. This is useful in simulation models where the spatial resolution of rasters changes through time, for example.

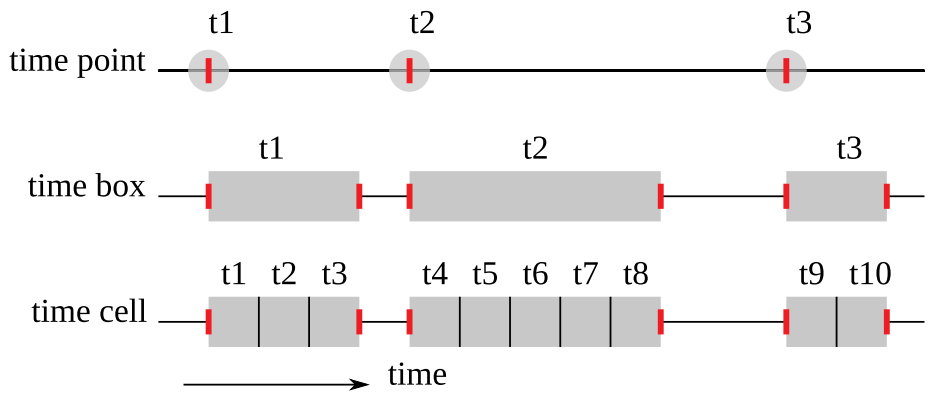


Fig. 10. Three approaches for representing locations in time that objects can be active: three time points, three time boxes, ten time cells. The red lines are the time points that are stored in the data model. For time cells, an additional count per time box is stored. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

3.3. Spatio - temporal objects (abstraction level 3)

Since we are now able to represent the individual kinds of object information (identity, locations in time, locations in space and properties), we can focus on the representation of actual spatio-temporal objects, in which all this information is combined and can be accessed in an integrated way. The abstractions mentioned in this section are part of the third and highest abstraction level implemented in our physical data model (Fig. 1).

One of the goals we had when designing the abstractions at this level was that it must be possible to coherently store object-related information at different (kinds of) locations in time and space. For example, for a collection of birds, it has to be possible to store information about the winter grounds, the migration route and the summer grounds. Another goal we had was that no information should be stored twice in the data model.

We identified two levels of grouping of object-related information. At the first level of grouping we combine information about which objects exist (identity), when they are active (locations in time), where each object is (locations in space) and what is present at those locations (properties). We call this level of grouping the property-set, the collection of locations in time the time domain, and the collection of locations in space the space domain. Within a property-set the information about the active sets, locations in space and properties is ordered by the locations in time. Also, within a property-set there can only be one time domain and one space domain. These domains are shared between all properties in the same set. Object-related information that is located at different locations in time or space is stored in different property-sets. For example, the constant sex of deer and their variable weight are stored in separate property-sets, having different time and space domains.

At the second and higher level of grouping, we group objects of the same kind with their property-sets. We call such a collection a phenomenon. Within a phenomenon, all objects are of the same kind, like deer or natural parks, and each object can be identified by a unique ID (identity). Also, all information related to a specific kind of objects is stored within a single phenomenon, in one or more property-sets.

This grouping of spatio-temporal object information is similar to the conceptual data model presented in de Bakker et al. (2017), whose design had a similar goal as our physical data model: to represent different kinds of state variables in a uniform manner in order to make it more convenient for environmental modellers to create models in which these different kinds of variables are manipulated. The conceptual data model is shown in Fig. 11. It defines *what* information is represented, and the physical data model defines *how* this information can be stored in a dataset. In an environmental modelling environment, the conceptual data model can be used as a basis for the representations, or data types, of the model state variables, whereas the physical data model can be used to allow these state variables to be persisted for later retrieval.

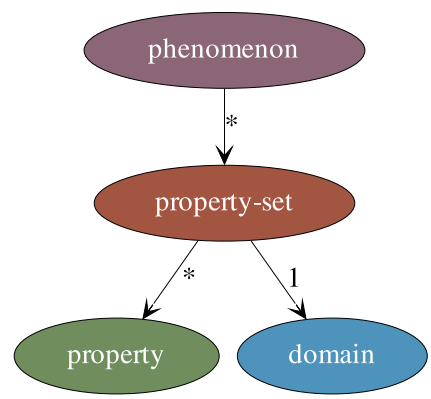


Fig. 11. Conceptual data model for representing different kinds of state variables (adapted from de Bakker et al. (2017)). A phenomenon contains zero or more property-sets, each of which contains a single domain (locations in time and space) and zero or more properties.

To illustrate the grouping of spatio-temporal object information in our data model, we will describe how to represent wandering deer. The implementation of this example is described in more detail in section 5. Fig. 12 shows the conceptual data model for a deer phenomenon. These deer are simulated using a model in which the spatial distribution of biomass in an area, and the location and weight of the deer are influenced by each other. The representation of the deer in our physical data model (Fig. 13) is very similar to the conceptual data model. For each deer the following information needs to be stored: a unique ID, the sex, the location in space and the weight. The ID and sex do not change through time, while the location in space and the weight do. Since within a property-set, there can only be a single combination of a time and space domain, this means that two property-sets are defined. In the first property-set, named constant, no locations in time and space are stored. For each deer, we store whether the deer is male or female in a property. In the second property-set, named variable, we use the time cell time domain kind, the mobile space point space domain kind, and a property for storing 0*D* numeric values representing the weights. Fig. 13 shows that all object-information is stored using one of the six array kinds described in section 3.1. In this case, most information is stored using the *variable value* × *same constant shape array* kinds, but this depends on the specific kind of information stored. When storing rasters for differently shaped areas, for example, a property containing *variable value* × *different constant shape array* values must be used. And when these rasters change shape through time, a property with *variable value* × *different variable shape array* values must be used, instead.

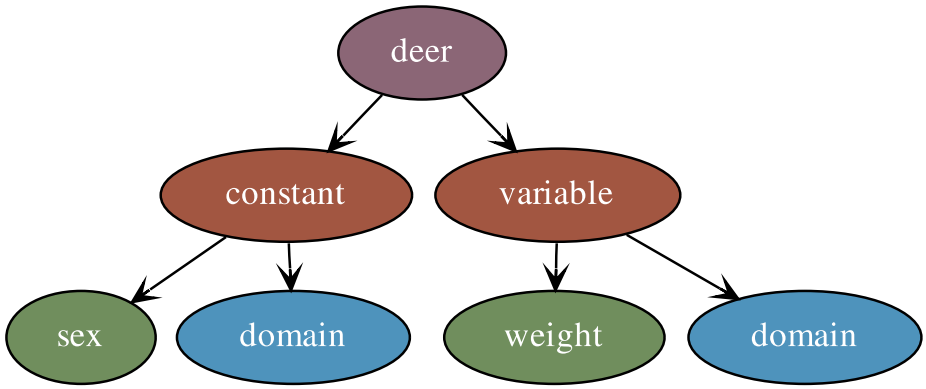


Fig. 12. The deer phenomenon represented by the conceptual data model of de Bakker et al. (2017). The colours correspond with those from Fig. 11. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

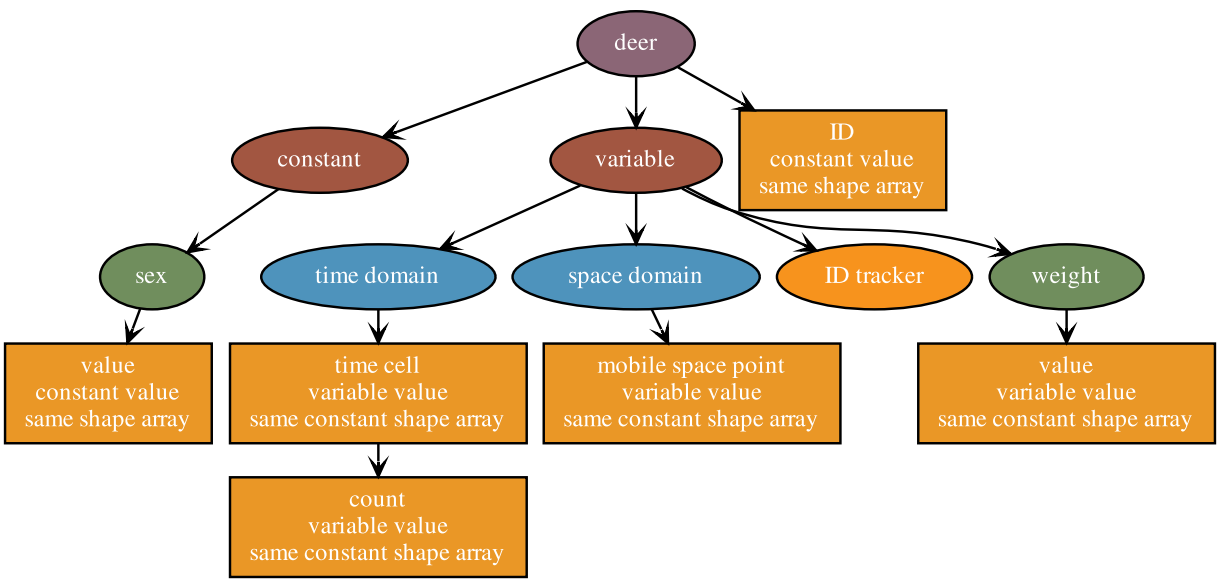


Fig. 13. Graphical representation of the physical data model of the deer phenomenon. Most colours correspond with those from Fig. 11. The orange boxes are collections of values represented by one of the six array kinds described in section 3.1. To keep the figure readable, the object ID tracker is not expanded into arrays. (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

4. Implementation

We implemented our physical data model using HDF5 (The HDF Group (1997–2019)). Besides being a software library and a file format, HDF5 is a data model itself. The HDF5 logical data model can be used to organize information to be stored in a file. The most important parts of the HDF5 data model are groups, datasets and attributes. Groups are used to aggregate other groups and datasets, while datasets are used for storing multidimensional arrays. With attributes, additional information can be associated with groups and datasets. The hierarchical nature of the data model (groups can contain other groups) allows for complex nested structures to be represented. Multiple HDF5 datasets (and groups and attributes) can be stored in a single HDF5 file. HDF5 datasets can be optionally configured to be extendable along one or more dimensions.

We have defined our data model in terms of the HDF5 logical data model, using mainly groups, datasets and attributes. The implementation, named LUE,[1] consists of a C++ library that implements the data model, and a Python package with which the data model is made accessible from Python scripts. In the C++ library we implemented the abstraction layers as discussed in section 3 separately (Fig. 1). We will now describe the implementation of each of these layers.

4.1. Arrays of temporal information

For each of the object array kinds described in section 3.1 and shown in Figs. 3–8, we have implemented code to store the value arrays. This low level of abstraction is implemented on top of a C++ library that wraps the HDF5 C library. For each object array kind, this layer contains code for reading and writing individual object arrays, or, when applicable, collections of object arrays. HDF5 datasets are used to store value arrays. It is possible to read or write a single 1*D* object array containing three floating point values, or multiple 1*D* object arrays in one go, each containing three floating point values, or such a collection of object arrays for multiple locations in time.

4.2. Spatio-temporal object information

For all kinds of object information as described in section 3.2, we developed code to read and write this information from and to a file. This code reuses the code mentioned in the previous section. Instead of

[1] LUE stands for Life, the Universe and Everything, mentioned in Douglas Adams' Hitchhiker's Guide to the Galaxy novels. Here, it refers to the fact that our data model is able to represent data about different kinds of information used in environmental modelling, and possibly other domains. We pronounce LUE as the French pronounce Louis (*LU-EE*).

reading and writing arrays of abstract temporal information, here object IDs, locations in time, locations in space, and properties are explicitly handled. The implementation contains higher level classes like MobileSpacePoint and Property which, in their implementation, use the lower level code for reading and writing arrays of temporal information. This makes it more convenient in higher level code to handle information related to spatio-temporal objects. For example, listing 1 shows the declaration of a function for creating a MobileSpacePoint instance. Such an instance manages the I/O of space points that move through space. The function is implemented in terms of the lower level API for handling arrays of temporal information. How this is done exactly is a detail that the caller does not have to be constantly aware of.

Listing 1: Subset of C++ API for creating mobile space points

4.3. Spatio-temporal objects

As described in section 3.3, the highest layer of abstraction of our physical data model contains mainly grouping constructs for aggregating information about the IDs, locations in time, locations in space, and properties of the objects. We have mapped these groups onto HDF5 groups. For example, listing 2 shows the declaration of a function for creating a PropertySet instance. Such an instance provides access to its object tracker, its time and space domains, and its properties.

Listing 2: Subset of C++ API for creating a property-set

It is possible to combine multiple phenomena into a single HDF5 file. This makes it possible to store all simulation state in a single LUE dataset.

4.4. Python package

In order to make it possible to manipulate LUE data sets from Python, we implemented a Python package on top of the higher abstraction levels of the C++ library. NumPy arrays (Oliphant (2006)) are used to interface object arrays between the Python and C++ code. This makes it possible to integrate the LUE physical data model with Python applications manipulating NumPy arrays, like SciPy (Jones et al. (2001)) and PCRaster (Karssenberg et al. (2010)). The pybind11 C++ library (Jakob et al. (2017)) is used to expose the relevant parts of the LUE C++ API to Python and vice versa. Examples of using the LUE Python package can be found in the next section.

5. Example: deer and biomass model

In this section we will describe how the state variables from a simple integrated agent-based and field-based model can be represented by our data model using the LUE Python package. We will highlight parts of the code. The complete model can be found in the source code repository associated with this manuscript (see section 8).

The purpose of the deer and biomass model used in this example is to illustrate that our data model is capable of storing both traditional agent-based and field-based model output. In this case the deer are the agents and the natural park area with a biomass raster is the field. Example output from the model is shown in Fig. 2. The model we used is simple and not realistic. The data is what we want to focus on here, not the validity of the model. It is possible to store more complex kinds of data than used here in our data model too, like properties with temporal discretizations, and phenomena with multiple time and space domains.

We realize that what we call a high level of abstraction can be considered a low level of abstraction by modellers. We envision that in real-world applications, an additional layer of abstraction will be built on top of our highest level of abstraction, making it easy for modellers to read and write model state information. This is shown as the upper layer of abstraction in Fig. 1. Concepts for how this can be done have been described in de Bakker et al. (2017).

The following examples assume that the LUE and NumPy Python packages have been imported, a dataset is created and two phenomena are added to it (listing 3). The returned Phenomenon instances (deer and park in this example) can be used to add property sets to the file.

After the model has finished executing, all model state variables have been stored in a single file, formatted according to the conventions described in this manuscript. Given that each LUE dataset is also an HDF5 file, the dataset can be copied to any other platform and the information stored can be retrieved again, using the LUE APIs and HDF5 APIs.

Listing 3: Import required packages

```
// Add mobile space point to the parent group and return
// a corresponding MobileSpacePoint instance.
MobileSpacePoint create_mobile_space_point(
    hdf5::Group& parent,
    hdf5::Datatype const& memory_datatype,
    std::size_t rank);

class MobileSpacePoint
{
    // Construct instance, based on parent group and
    // in-memory datatype of coordinates. The rank of
    // the space (1, 2, or 3) is read from file.
    MobileSpacePoint(
        hdf5::Group& parent,
        hdf5::Datatype const& memory_datatype);
};
```

```
// Add property-set to the parent group and return a
// corresponding instance. Information about the domain is
// used to setup the HDF5 constructs for writing locations
// in time and space.
PropertySet create_property_set(
    hdf5::Group& parent, std::string const& name,
    TimeConfiguration const& time_configuration,
    Clock const& clock,
    SpaceConfiguration const& space_configuration,
    hdf5::Datatype const& space_coordinate_datatype,
    std::size_t rank);

class MobileSpacePoint
{
    // Construct instance, based on parent group and name.
    // Information about the object tracker, time domain,
    // space domain and properties is read from file.
    PropertySet(
        hdf5::Group& parent,
        std::string const& name);
};
```

5.1. Initialize dataset

Before we can write the state of simulated spatio-temporal objects to a dataset, we must initialize the dataset. This will prepare the dataset for receiving state information.

In the deer and biomass model, the location and weight of each deer, and the distribution of biomass in the park within which the deer are located, change through time. As described in the previous text, time domains are contained in property-sets. Instead of storing multiple time domains with the same locations in time in multiple property-sets, they can be shared. This makes the resulting file smaller and prevents inconsistent time domains. In our example, the time domains of the property-sets containing temporal deer and park information can be shared. For this we create a separate phenomenon, called simulation, whose sole purpose is to contain a property-set with a time domain to be shared with other property-sets (listing 4). We could also have shared the time domain of a property-set in the deer phenomenon with a property-set in the park phenomenon, or vice-versa.

Listing 4: Create property-set with time cell domain and three-hourly time steps

The park phenomenon in our example model contains only a single object: the park within which all the deer are located. It contains a property-set named space_extent with a property named biomass for storing the 2*D* biomass arrays that change through time (listing 5). It also contains a property-set named constant with a property named space_discretization for storing the 1*D* space discretization information (number of rows and number of columns). The latter property is linked to the first to make explicit that biomass is discretized through space. The simulated biomass field is shown in Fig. 2b.

Listing 5: Create park phenomenon property-sets

The deer phenomenon contains for each deer the sex, the location in space and the weight. Preparing the LUE dataset for receiving state information during the simulation works similar to the previous case of the park phenomenon. In the case of deer the space domain is different (mobile space points are used instead of a stationary space box) and the weight property value has a different shape (0*D* instead of 2*D*). The simulated deer are shown in Fig. 2a. Fig. 14 shows the resulting state of the data model.

5.2. Write model state

Once the dataset is initialized we can start the simulation and write the state of simulated spatio-temporal objects to the dataset. In our case, this implies writing the initial state of the biomass raster of the park, and the sex, location and weight of each deer. After that, the interaction between the biomass field and the deer agents are simulated through time, and the biomass, location and weight of each deer are iteratively updated and written to the dataset.

In listing 6 the sex of each deer is written to the dataset. Given a phenomenon, property-sets can be obtained by name, and given those, properties can be obtained. Querying and assigning to property values works as if they are NumPy arrays. In this example, the value array is expanded to make space for an object array per deer. The expand method returns the value array on which it is called, which is then indexed for all object arrays. Assigning to these object arrays writes the new values to the dataset.

Listing 6: Write the sex of each deer to the dataset

```
import lue
import numpy as np

dataset = lue.create_dataset("deer.lue")
deer = dataset.add_phenomenon("deer")
park = dataset.add_phenomenon("park")
```

```
simulation = dataset.add_phenomenon("simulation")
epoch = lue.Epoch(lue.Epoch.Kind.common_era, "2019-01-01",
    lue.Calendar.gregorian)
clock = lue.Clock(epoch, lue.Unit.hour, 3)
time_extent = simulation.add_property_set("time_extent",
    lue.TimeConfiguration(lue.TimeDomainItemType.cell), clock)
```

```
space_extent = park.add_property_set("space_extent",
    time_extent.time_domain,
    lue.SpaceConfiguration(
        lue.Mobility.stationary, lue.SpaceDomainItemType.box),
    space_coordinate_dtype=space_coordinate_dtype, rank=2)
biomass = space_extent.add_property("biomass",
    dtype=np.dtype(np.float32), rank=2,
    shape_per_object=lue.ShapePerObject.different,
    shape_variability=lue.ShapeVariability.constant)
constant = park.add_property_set("constant")
space_discretization = constant.add_property(
    "space_discretization", dtype=lue.dtype.Count, shape=(2,))
biomass.set_space_discretization(
    lue.SpaceDiscretization.regular_grid, space_discretization)
```

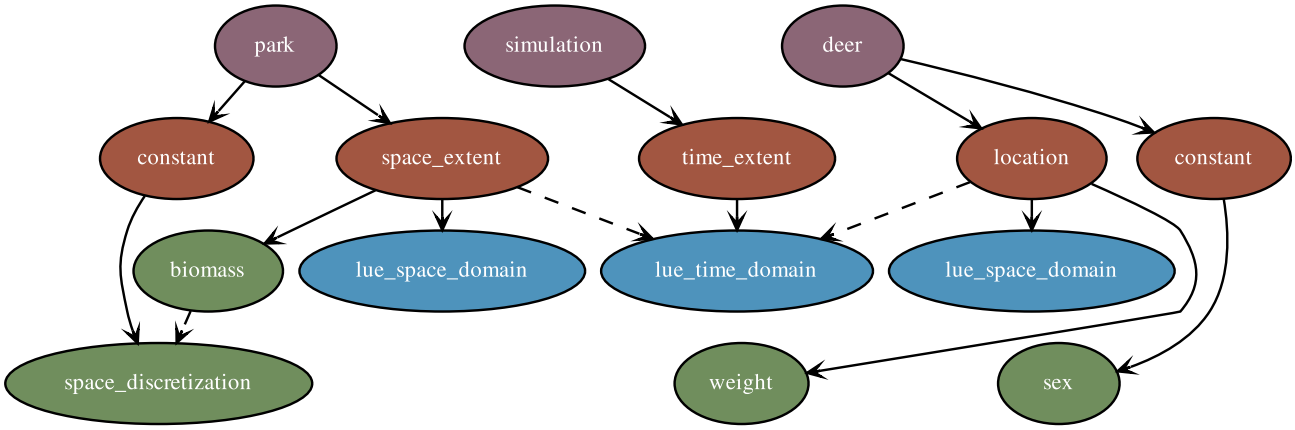


Fig. 14. Graphical representation of the physical data model of the deer and park phenomena. The dashed lines denote symbolic links to shared information. For clarity, the collections for tracking the object IDs are not shown.

Writing state that changes through time requires also writing the object IDs of the objects for which this state is written (the active set, AppendixA). Tracking the object IDs of the active deer is shown in listing 7. Mobile space points are represented by *variable value* × *same constant shape array* object arrays, which implies that for tracking the IDs of the objects the index of the active set and the collection of active object IDs must be stored.

Listing 7: Write the object IDs of the active deer to the dataset

Writing the updated location in space of active deer is shown in listing 8. After the new deer locations have been simulated, additional space is created in the dataset and their coordinates are written.

Listing 8: Write the locations of the active deer to the dataset

6. Discussion

We have designed and implemented a physical data model for simulated spatio-temporal objects. With this data model, information about the state of the simulated environment can be stored for post-processing. Although there are multiple approaches to represent state variables, depending on how they represent a value and the temporal and spatial extent and variability, we identified common aspects of these approaches, using a limited set of six kinds of object arrays (Figs. 3–8). These object arrays and the packing of them into value arrays form the basis of our data model. Layers with increasingly more domain-specific code are defined on top of this base layer, with the layer containing the support for spatio-temporal objects at the top (Fig. 1). Using this top-level layer of abstraction, we were able to represent some example kinds of state variables, suggesting that our data model is able to represent a diverse set of state variable kinds that otherwise would require multiple data models (section 5).

Our data model provides a unified approach to storing spatio-temporal objects. Using this data model, the environmental modeller can represent multiple kinds of state variables. The modeller is not forced to use a single data model for all state variables, or use multiple very different data models. In our view, this is the main contribution of this work.

In our data model, information about objects is stored in multiple arrays where each array contains a single kind of object-related information. This contrasts with many existing data models, where this information is stored in database table records. As mentioned in section 1, how information is organized in a physical data model might have an effect on how information is best organized in state variables in simulation models. Our data model suggests these state variables to be organized using arrays per kind of object-related information, with for the current modelled location in time the state of the current active sets of objects. This is different from the common approach in agent-based modelling of representing state variables by class instances aggregating all object-related information.

During this work we have identified some limitations of our data model, which have to do with usability and functionality. Our data model is capable of representing complex kinds of state variables, with changing collections of mobile objects and temporal discretizations of property values, for example. Although the data model is currently useable, we think it could be made more user-friendly for simple cases by adding an additional layer of abstraction, on top of the layer implementing the storage of spatio-temporal objects. This higher level data model would represent popular existing data models, like scalars, rasters and raster stacks, and time series, and serve as an easy to use replacement for existing data models. This would allow modellers to get familiar with the LUE data model without having to learn about the lower level details first, which are necessary for the more complex cases.

With respect to functionality, some aspects are still missing from the current version of our data model, that will be added at a later stage. Examples of these are the representation of relations between objects, uncertainty in temporal and spatial locations and property values, spatial projections, discretization of presence in space, other kinds of discretizations through space, more space domain item type like lines and polygons, spatial indices, and information about topology. Whether or not the data model has to be adjusted for these additional aspects to be added remains to be seen.

Although it is of crucial importance in a high-performance computing context, we have not evaluated the performance of our data model. Since our data model is implemented in terms of the HDF5 library, the performance characteristics of this library determine the performance of our data model. To reach maximum performance we must tune our use of HDF5 for our specific use case of storing spatio-temporal objects, and probably use parallel I/O on parallel file systems. This will be the focus of future research.

7. Conclusion

In this paper we showed the feasibility of a physical data model capable of storing different kinds of simulation model state variables in an integrated way. First, we defined spatio-temporal objects as uniquely identifiable objects for which locations in time, locations in space and properties are stored for those locations in time that the objects were active. By assuming that absence of information about presence in time or space just means that information is valid for all simulated time or space, we were able to frame all considered kinds of state variables as spatio-temporal objects, including simple state variable kinds, like scalars or non-temporal rasters.

Next, we presented our physical data model as consisting of multiple levels of abstraction. At the lowest level, arrays of general temporal arrays are represented. On top of that, spatio-temporal object information (object identity, locations in time and space, and properties) is represented, in terms of the general temporal arrays. At the highest level of abstraction, spatio-temporal objects are represented, as aggregates of spatio-temporal object information.

The resulting data model has been implemented in the LUE software package which contains APIs in C++ and Python. We have described an example of how LUE can be used from an integrated field-based and agent-based simulation model to store various kinds of state variables in a single dataset.

8. Software availability

The unified physical data model is implemented as part of a software

```
constant = deer.property_sets["constant"]
sex = constant.properties["sex"]
sex.value.expand(nr_deer)[:] = np.random.randint(
    low=0, high=2, size=nr_deer, dtype=np.uint8)
```

```
location = deer.property_sets["location"]
object_tracker = location.object_tracker
object_tracker.active_set_index.expand(1)[-1] = \
    object_tracker.active_set_index[-1] + nr_deer
object_tracker.active_object_id.expand(nr_deer)[-nr_deer:] = \
    deer_ids
```

```
deer_location = ...
location.space_domain.value.expand(nr_deer)[-nr_deer:] = \
    deer_location
```

package called LUE, which is hosted on Github at https://github.com/pcraster/lue. The data model is implemented by Kor de Jong in C++ and provides an API for C++ and Python clients. The code used for this work corresponds to Git tag 535d52b77092686e8d0641ae2e7983e5e9eddd3c and is freely available under the MIT open source license. A document called README.md is included in the root of the source code repository detailing the instructions for building the software. LUE is portable software and has been successfully built on various platforms (operating systems: Linux, macOS; compilers: Clang, GCC; architecture: *x*86-64). The README document also contains a link to the user documentation of the LUE Python package.

The example model used in section 5 can also be found on Github, at https://github.com/pcraster/paper_2019_physical_data_model.

Funding

This work was supported by the Research IT innovation programme (Utrecht University, The Netherlands) and the Global Geo Health Data Center (Utrecht University and University Medical Center Utrecht, The Netherlands).

Author contributions

KDJ and DK both worked on the concepts. KDJ designed and implemented the data model. KDJ wrote the manuscript, with inputs from DK. DK coordinated the project.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Object tracking

Here we describe in some more detail how we keep track of for which objects information is stored in the physical data model. This is relevant for the four object array kinds used to store temporal information. For each location in time and for each active object we store where each object array can be found. How this is done differs per object array kind.

Appendix A.1. variable value × same constant shape array

Per location in time, the value array a contains the values for all active objects (Fig. 5). An additional array *active_object_id* is used to store the IDs of the active objects (an ID is a unique number, see section 3.2.1). The order of information in a and *active_object_id* is the same. An additional array *active_set_index* is used to store the index into a and *active_object_id* where, for each location in time, the ID and object array of the first active object can be found. See also table A.2.

Table A.2
Procedures for storing and finding *variable value×same constant shape array* object arrays.

store	Append ID to *active_object_id* and object array to a
find	Given index of location in time, read index (s) of section of active objects from *active_set_index* and determine index (o) of object ID in *active_object_id*. The object array is located at $a[s + o]$.

Appendix A.2. variable value × same variable shape array

Per location in time, a separate value array a_t contains the values for all active objects (Fig. 7). Similar to the previous case, additional arrays *active_object_id* and *active_set_index* are used to track the IDs of active objects and the offset into *active_object_id* of the ID of the first active object, respectively. Each value array a_t is named after its corresponding (index of) location in time. See also table A.3.

Table A.3
Procedures for storing and finding *variable value×same variable shape array* object arrays.

store	Append ID to *active_object_id* and object array to a_t
find	Given index of location in time, read index of section of active objects from *active_set_index* and determine index o of object ID in *active_object_id*. The object array is located at $a_t[o]$.

Appendix A.3. variable value × different constant shape array

Per object, a separate value array a_o contains the values for all locations in time that the object was active (Fig. 6). Again, an additional array *active_object_id* is used to track the IDs of active objects. Array *active_object_index* is used to track the indices into the a_o arrays of the active objects. Array *active_set_index* is used to store the offset into *active_object_id* and *active_object_index* of the first active object per location in time. See also table A.4.

Table A.4
Procedures for storing and finding *variable value×different constant shape array* object arrays.

store	Append ID to *active_object_id*, append index of object array in a_o to *active_object_index*, and append object array to a_o
find	Given index of location in time, read index of section of active objects from *active_set_index* and determine index of object ID in *active_object_id*. Given this index, lookup index t of object array in *active_object_index*. The object array is located at $a_o[t]$.

Appendix A.4. variable value × different variable shape array

Per location in time and per object, a separate value array $a_{t,o}$ contains the object array (Fig. 8). See also table A.5.

Table A.5
Procedures for storing and finding *variable value×different variable shape array* object arrays.

store	Create value array named after current location in time and object, and write object array to it
find	Open value array named after current location in time and object

Appendix B. Representing time

In order to be able to represent time points of different resolutions (like nanoseconds and centuries), we used an approach inspired by the C++ chrono library.[2] Time points are dependent on a clock. A clock represents a period of time since an epoch, and has a certain resolution. This resolution is defined by the clock's tick period, which has a unit (seconds or years, for example) and a count (two for ticks of 2 s or years, for example). Given this, a time point can be represented by a duration, which is represented by a number of ticks.

We decided to represent a collection of time points by a single clock, represented by an epoch, a unit (a string) and a count (a positive integral number), and a duration for each time point, represented by counts (positive integral numbers). We represent epochs by an epoch-kind, and an optional origin and optional calendar (all strings). Currently supported epoch kinds are Common Era (CE) and Formation of Earth, but more can be added. The currently supported calendar is Gregorian.

Appendix C. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.envsoft.2019.104553.

References

UNIDATA, 2008–2018. Network common data form, version 4. https://www.unidata.ucar.edu/software/netcdf/docs/index.html. (Accessed 12 July 2019).
Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., Widmann, N., 1998. The multidimensional database system rasdaman. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data. SIGMOD '98. ACM, New York, NY, USA, pp. 575–577.
Becla, J., Zhang, D., Stonebraker, M., Brown, P., 05 2013. SciDB: a database management system for applications with complex analytics. Comput. Sci. Eng. 15, 54–62.
Bennett, D.A., Tang, W., 2006. Modelling adaptive, spatially aware, and mobile agents: Elk migration in Yellowstone. Int. J. Geogr. Inf. Sci. 20 (9), 1039–1066.
Castilla-Rho, J.C., Mariethoz, G., Rojas, R., Andersen, M.S., Kelly, B.F.J., 2015. An agent-based platform for simulating complex human-aquifer interactions in managed groundwater systems. Environ. Model. Softw 73, 305–323.
de Bakker, M.P., de Jong, K., Schmitz, O., Karssenberg, D., 2017. Design and demonstration of a data model to integrate agent-based and field-based modelling. Environ. Model. Softw 89, 172–189.
Fabian, R., 2018. Data-oriented Design: Software Engineering for Limited Resources and Short Schedules (Richard Fabian).
Filatova, T., Verburg, P.H., Parker, D.C., Stannard, C.A., 2013. Spatial agent-based models for socio-ecological systems: challenges and prospects. Environ. Model. Softw 45, 1–7.
GDAL/OGR contributors, 2019. GDAL/OGR Geospatial Data Abstraction Software Library. Open Source Geospatial Foundation. https://www.gdal.org. (Accessed 12 July 2019).
Goodchild, M.F., 2013. Prospects for a space-time GIS: space-time integration in geography and GIScience. Ann. Assoc. Am. Geogr. 103 (5), 1072–1077.
Jakob, W., Rhinelander, J., Moldovan, D., 2017. pybind11 – seamless operability between c++11 and python. https://github.com/pybind/pybind11. (Accessed 12 July 2019).
Jones, E., Oliphant, T., Peterson, P., et al., 2001. SciPy: open source scientific tools for Python. http://www.scipy.org/. (Accessed 12 July 2019).
Karssenberg, D., Schmitz, O., Salamon, P., de Jong, K., Bierkens, M.F.P., 2010. A software framework for construction of process-based stochastic spatio-temporal models and data assimilation. Environ. Model. Softw 25 (4), 489–502.
Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G., 2005. MASON: a multiagent simulation environment. Simulation 81 (7), 517–527.
Oliphant, T.E., 2006. A Guide to NumPy. Trelgol Publishing, USA.
Open Geospatial Consortium, 2011. OpenGIS Implementation Standard for Geographic Information - Simple Feature Access - Part 1: Common Architecture. Standard. Open

[2] http://en.cppreference.com/w/cpp/header/chrono.

Geospatial Consortium Inc. version 1.2.1. http://www.opengeospatial.org/standards/sfa. (Accessed 12 July 2019)
Parker, D.C., 2005. Integration of geographic information systems and agent-based models of land use: challenges and prospects. In: Maguire, D., Goodchild, M., Batty, M. (Eds.), GIS, Spatial Analysis and Modeling. ESRI press, Redlands, CA, USA, pp. 403–422.
Patterson, D., Hennessy, J., 2008. Computer Organization Design, fourth ed. Morgan Kaufmann.
PostGIS development community, 2019. PostGIS – spatial and geographic objects for PostgreSQL. https://postgis.net. (Accessed 12 July 2019).
Sbihi, H., Allen, R.W., Becker, A., Brook, J.R., Mandhane, P., Scott, J.A., Sears, M.R., Subbarao, P., Takaro, T.K., Turvey, S.E., Brauer, M., 2015. Perinatal exposure to traffic-related air pollution and atopy at 1 year of age in a multi-center Canadian birth cohort study. Environ. Health Perspect. 123 (9), 902–908.
Schelhaas, M.J., Kramer, K., Peltola, H., van der Werf, D.C., Wijdeven, S.M.J., 2007. Introducing tree interactions in wind damage simulation. Ecol. Model. 207 (2–4), 197–209.
Schippers, P., van Teeffelen, A.J.A., Verboom, J., Vos, C.C., Kramer, K., WallisDeVries, M.F., 2014. The impact of large herbivores on woodland-grassland dynamics in fragmented landscapes: the role of spatial configuration and disturbance. Ecol. Complex. 17, 13–20.
Sharp, J.A., Sep. 1980. Data oriented program design. ACM SIGPLAN Not. 15 (9), 44–57.
SQLite developers, 2000–2019. SQLite. https://www.sqlite.org. (Accessed 12 July 2019).
The HDF Group, 1997–2019. Hierarchical data format, version 5. http://www.hdfgroup.org/HDF5/. (Accessed 12 July 2019).
Wilensky, U., 1999. Netlogo. http://ccl.northwestern.edu/netlogo/. (Accessed 12 July 2019).