WISS Manual (Wageningen Integrated Systems Simulator)

DRAFT, DRAFT, DRAFT, DRAFT, DRAFT, DRAFT, DRAFT

Author: Daniel van Kraalingen, Rob Knapen, Hendrik Boogaard, Allard de Wit

Date : January 2020

1 Introduction

The WISS (Wageningen Integrated Systems Simulator) framework supports programming and operational application of simulation models in the agro-ecological modelling domain. WISS is written in the Java programming language and provides high numerical performance and robustness of the model implementations. To achieve these objectives, all simulation models in WISS are split into components that are essentially stateless and whose states and rates are managed by a dedicated state exchange object. Among other features, WISS provides facilities for runtime checking of state bounds as well as automatic scientific unit conversion allowing components that calculate in different physical units to seamlessly interact.

2 Ambitions

In developing WISS we strived for a software architecture that:

1) Is capable of calling a numerical model as a function. In other words the model should **not** have file based input **and** not have file based output. Simply setup the inputs, run the model and process the results of the model. While logging of information by WISS during model development is may seem a violation of this paradigm, the user can chose to run without logging and have completely 'silent execution.
2) Requires little or no changes to code describing scientific processes when this code is used in another model, or in another context.
3) Can convert, on the fly, values from one unit to another (e.g. degrees Celsius, to Kelvin or degrees Fahrenheit). This facilitates reuse of components and avoids having to make changes just for the sake of getting the units right.
4) Does automatic numeric bounds checking as the framework is running. This results in higher quality code of the model implementations, and more confidence in the final results.
5) Provides flexible coupling, starting and stopping of simulation components on the go. This enables more diverse compositions of simulation objects whereby e.g. one composition only runs a water balance of a bare soil under the prevailing weather conditions, and another composition adds a crop to that system.
6) Allows controlled data exchange between simulation components without requiring direct coupling of such components. This gives greater control of data flow and higher flexibility in composition of simulation components into a model.
7) Can externally force values of state variables. This enables the framework to override the outputs calculated by a simulation component with e.g. a measured value.
8) Keeps states of simulation and integration of those states centralized and not dispersed throughout multiple model components. Enabling querying of the model outputs after model termination, or persist / reinstate the state of the simulation, among other advantages.
9) Provides excellent performance. For regional and global agro-ecological studies (with ever higher spatial resolutions), superfast performance is crucial to reducing computing costs.
10) Can run on as many hard and software platforms as possible (Windows, Linux, macOS).

3 Principles and terminology of WISS

This chapter gives an overview of the basic principles and terminology of WISS that a modeller, who plans to construct a crop model, needs to understand.

3.1 Java implementation

WISS is currently implemented in the Java language. Software written in Java can run on a large number of platforms, such as Microsoft Windows, macOS, Linux (see www.java.com/en/download/help/sysreq.xml for details). Java is intended to let application developers "write once, run anywhere", meaning that compiled Java code can run unchanged on all Java supported platforms **without** the need for **recompilation** (a large advantage compared to Fortran, C, C++ and other languages that require the user to recompile the code with the appropriate compiler installed on the target platform).

To achieve that most Java compilers produce platform-neutral Java bytecode. A Java Virtual Machine (JVM) then takes this bytecode and does just-in-time compilation into native machine code to run on the target platform. Hence, to use WISS a Java Virtual Machine needs to be installed. JVMs are available as part of a JDK (Java Developer Kit), and a JRE (Java Runtime Environment), both are freely available for many operating systems, if not already installed on the machine you are working on. Try running the command:

```
Java<enter>
```

For more information please see openjdk.java.net and www.java.com. Lately there are also JDKs being made available that are optimized for running Java programs in cloud environments, such as www.eclipse.org/openj9/ and aws.amazon.com/corretto/.

WISS could be translated into other object oriented languages if necessary.

3.2 What is a WISS model?

In essence a WISS model is a class derived from the Model class in which one or more SimControllers are started. The SimControllers are the actual controller objects starting monitoring and stopping the simulating objects: SimObjects. The SimControllers actually determine what the model should do! See below for explanations of Model, SimControllers and SimObjects.

Experience in Java or some other object oriented programming is helpful in understanding and working with WISS!

3.3 Daily time steps with rectangular integration

WISS is targeted at the agro-ecological modelling domain. In this domain, daily time steps in numerical integration are most common (basically driven by the nature of daily weather data, hourly being much harder to get). Systems with flexible (=variable) time steps tend to introduce a level of complexity that would make programming a WISS model much less straightforward, and would slow down development of the initial framework. Simulation with flexible time steps may be introduced in future versions, should the need arise.

Concurrent with daily time steps, rectangular integration (Euler) is the provided integration method.

3.4 Parameters and initial states (ParXChange)

Xxxx nog doen opmerking hendrik in comments over verschil tussen parxchange en simxchange

Parameters and initial states in WISS are not read from file / keyboard / database etc. in the code sections where calculations are done. A principle of WISS is that all input data necessary to make a calculation is provided by a special object, so the calculation sections have no way of 'knowing' where this data comes from.

Provision of input data (parameters, initial states etc.) in WISS is by **one** instantiated ParXChange object. This ParXChange object must be provided with the right input data before start of simulation.

The ParXChange object can deliver data of many types (whole numbers, floating point numbers, interpolation tables etc.) to a caller in a different unit from how it was provided, and will give a blocking error when a parameter value is requested that was not set earlier.

3.5 Integration and shared state approach (SimXChange)

In crop modelling there has been a tendency over the years for more modularization and looser coupling of software code describing the principal physiological and physical processes. As model descriptions became more complex and understanding of reality improved, software architecture principles have been adopted to increase modularization. But modularization also required solutions for inter process communication of data. In some implementations large arguments lists were moved around, where in some cases people could hit the maximum of 255 arguments in some Fortran implementations. In other cases, large blocks of global storage were used (such as 'common blocks' in Fortran). Other solutions tended to introduce a software component responsible for exchange of state and other variables. In all such solutions, ownership of state variables is (of course) with the software code where the states are calculated, but basically copies of these variables are communicated and are supplied on request by the communication component to other parts of the model, providing some greater flexibility and protection at the expense of model performance.

In WISS we've taken a different, new approach. States in WISS are not kept permanently in software code describing model processes but are kept in **one** instantiated SimXChange object which manages and stores all states since the start of the simulation and through which all communication between simulation objects takes place. This approach we call the "shared state" approach. Even the owned states of a simulation object are maintained in this object, and are obtained at every time step prior to calculation.

States in this object are registered such that other simulation objects can only read the value and cannot accidentally change the value. This is done by returning a 'token' (=special identifier) on registration of a state variable. This token contains the privilege whether the holder can change the state value (by providing a rate value), or can only read the state value. This token also enables fast storage and retrieval of data as it contains the direct location of the data in the SimChange object, i.e. searching for the data is not necessary. It is this principle that facilitates the high performance of a WISS model.

States that are registered in SimXChange are given a valid range on registration. While simulation is running, the simulation object provides the required unit for each get and set. Meanwhile, the framework checks to see whether registered ranges are met and illegal unit conversions are prohibited.

It is in this object that numerical integration takes place whenever time of simulation steps to the next simulation time.

Conclusion: basically all time-related communication of data among simulation objects is done through a special object also called the simulation exchange object, or SimXChange object. As this object still exists after termination of the simulation, it can be queried programmatically for data gathered during simulation. Methods exist to obtain single values on a simulation date for a variable, time series containing all dates and values of a variable and aggregated data of a variable (such as the first, last, highest and lowest values, first date of entry, last date of entry etc.).

3.6 State, driving, auxiliary and external time dependent variables

State variables are variables determining the state of the system, as is discussed above they are not integrated in the simulation objects where calculations take place but are integrated in the provided SimXChange object immediately after the time step is set to the next day. Consequently, rates of change must be published to the provided SimXChange object.

An auxiliary variable is a time dependent quantity, but not being a state ! Semantically one could distinguish between time dependent quantities not influenced by the system (e.g. driving variables such as incoming radiation) and time dependent quantities that are influenced by the system (real auxiliary variables, e.g. daily crop transpiration). A distinction is made between the two in the simulation object that publish them (one is controlled by a rate calculation, the other can just be published). However, to a simulation object needing an external time dependent quantity, no distinction is necessary between a state from another simulation object, a driving variable from somewhere, or a real auxiliary variable from somewhere. These can be requested from SimXChange simply as an **external variable**.

3.7 Separation of calculations (SimObjects and SimIDs)

Scientific calculations can be separated into different 'units', which, obviously must interact during simulation with each other by exchanging data. In WISS these units of calculation are generically called **SimObjects** and they must provide implementations for abstract methods inherited from the parent class SimObject. So e.g. SimAstro is a class that inherits from SimObject, designed to provide daylength calculations on the current simulation date, SimMeteo is a class that inherits from SimObject, designed to provide meteo elements on the current simulation date. In general the SimAstro, SimMeteo and other implementations are meant whenever reference is made to 'SimObjects'. So the basic responsibility of SimObjects is to provide calculation results to SimXChange once instantiated.

A particular SimObject implementation (e.g. SimAstro) does nothing really if it is not instantiated (a basic principle of Java and other Object Oriented languages). To be able to distinguish SimObjects from each other during simulation and in the generated output, a SimObject receives a unique string identifier from the SimController starting it (a so called SimID) on instantiation that the running object must use in its communication with the 'outside world'.

A simulation object can have one or more state and/or auxiliary variables but does not have to, although a simulation object with no states and no auxiliary variables is probably trivial.

Upon instantiation of a SimObject it must obtain parameters and initial states from the provided ParXChange object.

A principle of WISS is that a SimObject does not do state integration itself but defers this to the provided SimXChange object. Therefore, the part of a SimObject that deals with real calculations must request its own state variables and external variables (other state and auxiliary variables) from the provided SimXChange object, prior to any numerical calculation. After calculations, it must push its calculated variables to the provided SimXChange object. The token mechanism guarantees that there is only a minimal performance penalty for doing so.

A simulation object must consist of a number of basic separate calculation units (called methods in Java), each suited for a particular task.

These methods are:

1) Construction:
   Registering it's SimID with SimXChange, registering owned state variables and owned auxiliary variables (not being a state) with SimXChange, getting parameters and initial state values from ParXChange and setting initial state values in SimXChange.
2) Intervene:
   Optional overriding and publishing owned state variables to the provided SimXChange object.
3) Auxiliary calculations:
   Optional calculation and publishing of owned auxiliary variables (not being a state) to the provided SimXChange object.
4) Rate calculations:
   Optional pulling required states and auxiliary variables from SimXChange, doing calculations, publishing rates of change to the provided SimXChange object.
5) CanContinue:
   Optional boolean function whereby the SimObject can indicate that it cannot continue. If false, the SimObject will be terminated.
6) Terminate:
   Optional any stuff that the SimObject needs to do prior to object destruction.

3.8 SimXChange internal data structure after registration and simulation

We will elaborate somewhat here in order to better understand the mutual relations between SimObjects, SimIDs variable names and tokens by means of an example where a fictitious crop rotation is done on a permanent soil. The described example system is one where SimObjectSoil is instantiated and registered by the simulation controller with SimID=SimObjectSoil and publishes X1 and X2 from the start until the end. A crop objec (SimObjectCrop) is instantiated and registered by the simulation controller on the second day of simulation with SimID=SimObjectCrop_1 and runs for two days publishing Y1 and Y2. On the fourth day, SimObjectCrop is instantiated **again** and registered by the simulation controller with SimID=SimObjectCrop_2 and run again now for three days (remember it is an example crop rotation). N.B. A registration with a particular SimID can take place only once, so instances of the same SimObject must be registered with different SimIDs (for good reason). Examples for write enabled and read-only tokens are given below.

The situation in SimXChange on the first date of simulation (date=0) is given below:

				startdate							enddate
SimID	VarName	Token		0	1	2	3	4	5	6	7
SimObjectSoil	X1	6916/53381		value							
SimObjectSoil	X2	5700/67904		value							

Note that only one SimObject with SimID=SimObjectSoil is registered, providing initial values for X1 and X2.

The situation in SimXChange on date=1 is given below:

				startdate							enddate
SimID	VarName	Token		0	1	2	3	4	5	6	7
SimObjectSoil	X1	6916/53381		value	value						
SimObjectSoil	X2	5700/67904		value	value						
SimObjectCrop_1	Y1	8450/15899			value						
SimObjectCrop_1	Y2	2073/05447			value						

Note that there is now a new SimObject registered with SimID=SimObjectCrop_1 and it has provided initial values for Y1 and Y2.

The situation in SimXChange on date=3 is given below:

			startdate							enddate
SimID	VarName	Token	0	1	2	3	4	5	6	7
SimObjectSoil	X1	6916/53381	value	value	value	value				
SimObjectSoil	X2	5700/67904	value	value	value	value				
SimObjectCrop_1	Y1	8450/15899		value	value					
SimObjectCrop_1	Y2	2073/05447		value	value					

Note that on date=2, values were provided for X1, X2, Y1 and Y2. However, date=2 was the last simulation date of the SimObject that has SimID=SimObjectCrop_1, so values Y1 and Y2 are missing again on date=3.

The situation in SimXChange on date=4 is given below:

			startdate							enddate
SimID	VarName	Token	0	1	2	3	4	5	6	7
SimObjectSoil	X1	6916/53381	value	value	value	value	value			
SimObjectSoil	X2	5700/67904	value	value	value	value	value			
SimObjectCrop_1	Y1	8450/15899		value	value					
SimObjectCrop_1	Y2	2073/05447		value	value					
SimObjectCrop_2	Y1	6181/63640					value			
SimObjectCrop_2	Y2	9498/84746					value			

Note that on date=4 a SimObject got registered with SimID=SimObjectCrop_2, providing also Y1 and Y2 with initial values, but with a different SimID.

The situation on the end date is a follows:

			startdate							enddate
SimID	VarName	Token	0	1	2	3	4	5	6	7
SimObjectSoil	X1	6916/53381	value	value	value	value	value	value	value	value
SimObjectSoil	X2	5700/67904	value	value	value	value	value	value	value	value
SimObjectCrop_1	Y1	8450/15899		value	value					
SimObjectCrop_1	Y2	2073/05447		value	value					
SimObjectCrop_2	Y1	6181/63640					value	value	value	
SimObjectCrop_2	Y2	9498/84746					value	value	value	

Also the SimObject with SimID=SimObjectCrop_2 has now terminated (already on date=6). Simulation has reached its end date and SimObjectSoii is also terminated.

It is important to understand that when a particular SimObject requests Y1 from SimXChange it needs to anticipate on the possibility that it is not available and must act accordingly by using its own default value. It is advised to document this properly and even show this through logging if non trivial defaults are used. A trivial case which defaults to a zero value would one where a SimObject requests a leaf area index which is not available. Also important to understand is that the values for Y1 and Y2 on date=1 and 2 will come from the SimObject registered with SimID=SimObjectCrop_1 whereas the values for Y1 and Y2 on date=4, 5 and 6 will come from the SimObject registered with SimID=SimObjectCrop_2!

Important to note is that a particular variable cannot be "switched on" again after a previous period with valid values by the same SimID. A variable does not have to be present from the first date of simulation, obviously because the SimObject does not have to be started from the first date of simulation, but if once during a time step a new value was not provided it will be switched off by SimXChange, trying to switch it on again by the same SimID will result in an exception error. Having the same variable name again in SimXChange can only be from a registration with another SimID.

3.9 Separation of control (SimControllers)

A special type of object is available in WISS which can **monitor** the simulation during its entirety and start, stop or otherwise interact with the scientific calculations as defined in SimObjects. This type of object is called a SimController and this is the place where SimObjects are instantiated, monitored and terminated. SimControllers typically start the SimObjects dealing with model environment (e.g. the order of the different calculations such as weather, soil moisture), as well as deal with any type of intervention on the running simulation e.g. intermediate and final harvesting. The basic responsibility of a SimController being to instantiate (start) a SimObject of the right type, or stop a SimObject once a termination criterion is reached. A model in WISS can have one or more SimControllers, one can deal specifically with instantiation of SimObjects while another can deal specifically with termination of SimObjects. Basically any number of SimControllers can observe the ongoing simulation and interact with it.

It is the SimController's responsibility to provide a proper SimID to the instantiated SimObject. If, during one simulation run, there is only one instantiated object per SimObject class (e.g. when only one SimAstro is instantiated), the provided SimID could be just the SimObject's class name, since a clash of names will not occur, but from two or more objects from the same SimObject's class during a simulation run (e.g. when a crop rotation of the same crop (e.g. SimCrop) is calculated), it is required to give each SimObject's instantiation a unique SimID.

Usually when a SimController is required in a particular WISS model, it will be instantiated upon start of the simulation, otherwise it cannot carry out its monitoring task while time is progressing. In fact how a WISS model will 'behave' will be largely determined by which SimController(s) are started, and their implementation.

In general, a WISS model with no instantiated SimController will do nothing, since no SimObject will be started. Time will simply progress until the end date and no results will be present in SimXChange.

3.10 Run-time unit conversion (ParXChange, SimXChange)

Any numeric data that is exchanged among SimObjects and SimControllers is exchanged using 2 specialized objects (for input data, the ParXChange object, for simulation data, the SimXChange object, see previous sections). The unit of the numerical values is stored on first entry and used to convert should a caller want to have the value in a different unit. An exception error will occur if the unit conversion has not been defined. Not all valid unit conversions have been programmed in WISS yet, so it could well be that the required unit conversion needs to be added to the WISS framework.

Unit conversion is done in the class ScientificUnitConversion, although the modeller will probably never deal with this class directly.

3.11 Run-time bounds checking (SimXChange)

In the object for dynamic data exchange (SimXChange), valid bounds are given on initial registration of state variables. So whenever a WISS simulation object defines an owned state variable or a variable that it is going to 'publish', making it available to other simulation objects, it can define the value range in

which it must lie. The dynamic SimXChange object will on every publication of the variable, check the new incoming value against the value range and throw an exception whenever this is violated.

3.12 Exchange of data (SimXChange)

Exchange of data through SimXChange can take place in several different ways. There is the basic level in which a simulating SimObject registers its own states with SimXChange using the state name and provided SimID, receiving a token (coding for SimID **and** state name) through which rates of changes can be sent to SimXChange and through which the numeric state value can be obtained from SimXChange for calculation purposes. This functionality would allow the SimObject to run if external states are not needed for the calculations.

A different way of obtaining data is when a SimObject needs an external variable, but of course it does not want to have knowledge of the publisher. In other words it just needs the value of the external variable, by name only, independent of which SimObject owns it, so independent of the SimID !! The SimXChange object provides functionality for this. At every external variable request the SimXChange object will return the value if there is a SimObject publishing this state on the simulation date (there can be at most one SimObject publishing it). A missing value will be returned if there is no publisher. The SimObject requesting it can then take action and use a default instead.

Consider a case where there is a water balance submodel and a crop submodel. If the soil water balance is running it will probably need info on the crop cover, rooting depth, extraction of water by roots, etcetera. If crop cover and the other variables are not available, the soil water balance could use zero for these external variables.

Consider another case where e.g. a soil water balance first has a crop A, then crop B, after a fallow period (i.e. a crop rotation). In such a case the crop cover and other variables are provided at some point in time by another SimObject (e.g. of SimCropB, instead of from SimCropA). From the point of view of the soil water balance, SimXChange allows for a seamless switch from info provided by crop A, to absence of info, to info provided by crop B.

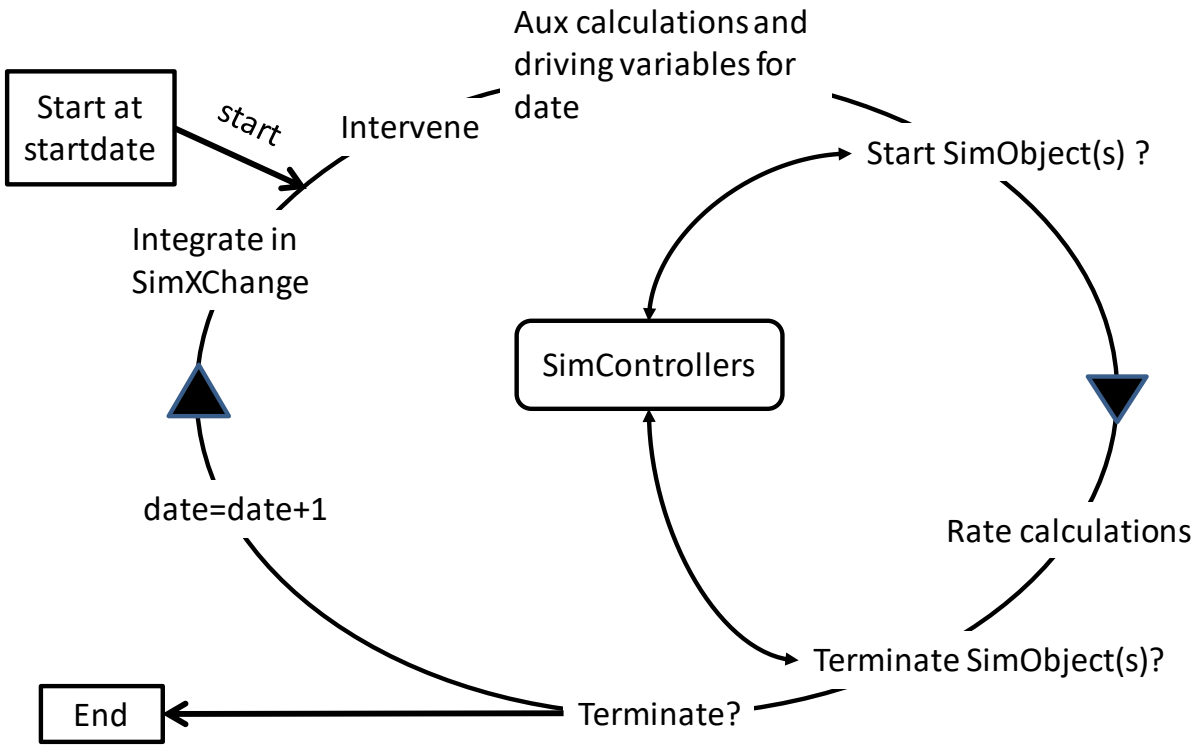
This mechanism can only work if the names of states (and other types of data), are determined globally. Therefore in every SimObject you'll see names originating from a global list of names defined in the Model subclass.

3.13 Model composition (Model)

The composition of SimControllers that will be used in a specific model situation is determined in a Model derived class which instantiates the required SimControllers to calculate something sensible. Together with parameterization, instantiated SimControllers and available SimObjects, this is what determines what the WISS model will calculate.

3.14 Time loop (TimeDriver)

An essential part of any dynamic simulation model is the order in which calculation steps and date progression are carried out.



The drawing above shows the sequence of events that take place in WISS while the model is running. The implementation in code is in the TimeDriver class. Execution is in clockwise order, starting from "Start at start date". The following steps are made:

- Start one or more SimControllers (depending on the model, not shown here for clarity). These will oversee the simulation and start and stop the SimObject(s) if necessary (but starting is actually done later in the loop).
- Intervene:
  The Intervene step is an opportunity for every running SimObject in the system to override (=force) any state variable in the provided SimXChange object. Here adding or taking away part or whole of a state variable is allowed (if within the registered bounds, and the consistency of the model is retained). The provided SimXChange object will report these forcings in a special report with the date of overriding, the old and the new value.
- Aux calculations:
  All running SimObjects are called to obtain (if required) data from the provided SimXChange object to calculate and provide auxiliary variables back to the provided SimXChange object. After this point, all existing states and external variables are up to date for the current simulation date!
- Start SimObject(s) ?:
  All running SimControllers are asked whether additional SimObject(s) must be started. All started SimObject(s) also have their AuxCalculations method called (not shown here). **At this point the system is up to date for the new states and auxiliary variables for the current simulation date.**
- Rate calculations:
  All running SimObjects are called to carry out rate calculations and publish those to SimXChange.
- Terminate SimObjects(s):
  Call all controllers and let them terminate the SimObject(s) which need to be terminated (through evaluation of states etc.), also ask remaining running SimObject(s) whether they can continue (method: SimObject.canContinue), if not, terminate them too.

- Terminate?:
  Terminate time loop if there were any previous running SimObjects but there are none anymore, or the simulation's end date has been reached. If either of that happens, any remaining running SimObject(s) are terminated, the time loop ends, and post processing can take place.
- Date=Date+1:
  Date increase if the loop is not terminated.
- Integrate in SimXChange:
  The internal states of SimXChange are integrated only if a rate has been provided. If the rate of change for a state has not been provided, the state will be missing from the new date until the end date. Auxiliary variables on the new date all have a NaN value, waiting to be set in the AuxCalculations step.

3.15 Errors in WISS

WISS is designed to safeguard valid simulation as much as possible. In general simulation will be terminated by a run-time exception whenever something goes wrong. Beit a non-existing unit conversion, a bounds check error, a required external variable not there etcetera. The principle being that if an error occurs, simulation results are unreliable anyway, so there is no need to continue. Best is to present the error with as much information as possible to the user, so the error can be located easily and repaired.

In Java, contrary to other computer languages, floating point exceptions such as divisions by zero and positive and negative infinity do not automatically result in run time exceptions. There are special sentinel values for such situations which are handled in any successive calculations. This is considered an undesirably situation for model development because the model results are unreliable anyway after such events, so it is best to stop **immediately** on such occasions. For that purpose there is a function in the WISS framework (safeExpr, class RangeUtils), and you are advised to use it whenever there is a likelihood of division by zero or some kind of infinity. This function will immediately throw an exception in these cases. Division by zero in Integer values does cause an immediate exception.

3.16 Logging

WISS provides different levels of logging (through Apache Log4J (logging.apache.org/log4j/2.x/). These levels are:

- Error:
  Logging of Exception messages (are always fatal in WISS).
- Warn:
  Logging of warnings for unused input parameters.
- Info:
  Logging of basic info on the run before running.
- Debug:
  Logging of start and termination date and name of SimObjects and SimControllers.
- Trace:
  Logging of info all variables published to SimXChange and method calls of all running SimObjects. To avoid the performance penalty from providing the logging object with trace info all the time, trace logging to the logging object is actually only done when the boolean TRACELOGGING is set to true in the provided ParXChange object.

4 My first WISS model

In this section we will discuss a WISS implementation of the popular prey-predator (Lotka-Volterra) model, well known from population dynamics as an example.

When developing a new model using WISS, one has to analyse the system in terms of separate processes (i.e. simulation objects, these go into SimObject implementations). However, SimObjects in WISS do nothing but wait to be instantiated by a SimController implementation which starts, runs and stops the SimObject implementations in the proper order. Thus for our new model we must also provide at least one Controller implementation. Note that we can have different SimController implementations each running the same SimObject implementations, so we can organise different models with the same SimObject implementations! Finally, to do a real simulation run we must provide initial conditions and model and time parameters.

In summary we have to provide:

- At least one SimObject implementation (describing the system numerically)
- At least one SimController implementation (describing starting and optional stopping)
- Exactly one Model implementation
- A set of input parameters and initial states

The system that we will be modelling here is described by:

```
FOUND = C * (1.0 - EXP (-1.0 * A * PREY / C))
RPREY = R * PREY * (1.0 - PREY/K) - FOUND * PRED
RPRED = (B * FOUND - D) * PRED
```

Where RPREY is the rate of change of the PREY state (the number of prey animals), and RPRED is the rate of change of the PRED state (the number of predator animals). The system has parameters A, B, C, D, K and R.

In the case of the prey-predator model we separate this system into two SimObject implementations: called SimPrey and SimPredator. The name for our standard controller is SimControllerModelPreyPredStandard, while the Model implementation is called ModelPreyPred1, where "1" signifies that we could have more than one implementation each having their own characteristics.

4.1 The SimObject implementations

When implementing a SimObject one has to analyse the system in terms of the following aspects:

1) What are the state variables that the SimObject implementation must calculate ?
   Each state requires a rate variable to be calculated within the rate calculation section as well as an initial value, to be obtained on instantiation from the provided ParXChange object (or zero in special situations). Also the unit for each state variable has to be determined. Communication of state and rate variables has to done to the provided SimXChange object.
2) What are the parameters required for the calculations ?
   Each parameter is obtained on instantiation of the SimObject implementation from the provided ParXChange object.
3) What are external time dependent state and external auxiliary variables required for the calculations ?
   These are obtained **while running** from the provided SimXChange object.
4) Is there a condition on which the SimObject implementation wants to be stopped ?

For SimPrey somewhat abbreviated Java code is given below. This won't run as given here (the real code would take up too much horizontal and vertical space), but is meant to explain the essential workings.

Code for SimPredator is not discussed here as it would lead to a lot of duplication, see the example files for it in directory xxxx.

The class declaration and field definition section pseudocode:

```
public class SimPrey extends SimObject {
    double a = parXChange.get("A", ScientificUnit.NA);
    double c = parXChange.get("C", ScientificUnit.NA);
    double k = parXChange.get("K", ScientificUnit.NA);
    double r = parXChange.get("R", ScientificUnit.NA);

    SimValueExternal pred = new SimValueExternal("PREDSTATE", ScientificUnit.CNT_HA);

    SimValueState prey = new SimValueState("PREYSTATE", ScientificUnit.CNT_HA,
                                           RangeUtils.RangeType.ZEROPOSITIVE);
}
```

The class declaration and field definition section explained:

- The parameters required for calculation of the prey rate (A, C, K, R) are obtained from the provided parXChange object. We ignore the scientific unit for these parameters (hence ScientificUnit.NA).
- An object "pred" with label "PREDSTATE" is created of type SimValueExternal for obtaining values for the external predator state. The unit in which we want to use that state is the number of animals per hectare (values are obtained during actual simulation, this section only creates the object to hold that info). A SimValueExternal object has properties such as the value, the unit of the value etc.. Note that for external values no difference needs to be made between states and auxiliary variables. They are all the same to a SimObject.
- An object "prey" with label "PREYSTATE" is created of type SimValueState for holding prey rate and state calculation results, the unit we will use also is the number of animals per hectare, and the range of the state can be anything from zero to infinity (negative values are prohibited, and will throw an exception error). A SimValueState object has properties such as the state and rate value, the unit etc.

The object constructor pseudocode:

```
public SimPrey(String     aSimID,
               ParXChange aParXChange,
               SimXChange aSimXChange) {
    super(aSimID, aParXChange, aSimXChange);

    prey.v = parXChange.get("PREYSTATE", simID, ScientificUnit.CNT_HA);

    this.pushSimXChangeData();
    this.auxCalculations();
    }
```

The object constructor explained:

- The simulation id (aSimID), ParXChange and SimXChange object to be used by this object's instance are received from the caller and passed on to private fields so these are available in every method of this object.
- The initial prey number is obtained from the provided ParXChange object and assigned to the value property (.v) of the prey object.

- Call the pushSimXChangeData method to push calculated data to the provided SimXChange object (in this case only the initial value of the number of preys). This method is explained below.
- Always call auxCalculations as last line in a constructor (**after** this.pushSimXChangeData), guarantees that all variables that the SimObject is responsible for are actually published during construction.

The rate calculations pseudocode:

```
public void rateCalculations() {
    this.pullSimXChangeData();

    double found = c * (1.0 - Math.exp(-1.0 * a * prey.v / c));
    prey.r = r * prey.v * (1.0 - prey.v / k) - found * pred.v();

    this.pushSimXChangeData();
}
```

The rate calculations explained:

- Call the pullSimXChangeData method to pull (get) the required data from the provided SimXChange object. This method is explained below. Note that also owned states are obtained runtime from the provided SimXChange object. They must not be kept local, otherwise external forcing wouldn't be possible.
- Calculate the prey rate of change and assign to prey.r.
- Call the pushSimXChangeData method to push calculated data to the provided SimXChange object (in this case rate of change of the number of preys). This method is explained below.

The pullSimXChangeData pseudocode:

```
public void pullSimXChangeData() {

    // get copies of all external variables (<SimValueExternal> variables) that need to
    // be there at any time during object existence, use:
    // simXChange.getSimValueExternalByVarName(<SimValueExternal>)

    simXChange.getSimValueExternalByVarName(pred);

    if (this.isIntervening() ||
        this.isAuxCalculating() ||
        this.isRateCalculating()) {

        // get copies of all owned states (<SimValueState> variables),
        // only when in time loop, use:
        // simXChange.getSimValueState(<SimValueState>)

        simXChange.getSimValueState(prey);
    }
}
```

The pullSimXChangeData explained:

- This method is meant to get required data from the provided SimXChange object.

- Try to get data from the provided SimXChange object for the external predator number through pred. A NaN is returned if pred cannot be found in the provided SimXChange object. Note that a NaN will immediately lead to an exception when the value (.v()) is used in calculations.
- Get the prey number state from the provided SimXChange object, only for certain calculations. An error will occur if this cannot be found.
- Every variable to be pulled from SimXChange must be mentioned in this section !

The pushSimXChangeData pseudocode:

```
public void pushSimXChangeData() {

    if (this.isInitializing()) {
        // set initial value of all owned states (<SimValueState> variables), use:
        // simXChange.forceSimValueState(<SimValueState>)

        simXChange.forceSimValueState(prey);

    } else if (this.isAuxCalculating()) {

        // set replacement values of all owned auxiliary variables
        // (<SimValueAux> variables), use:
        // simXChange.setSimValueAux(<SimValueAux>)

    } else if (this.isRateCalculating()) {
        // set rates of all owned states (<SimValueState> variables), use:
        // simXChange.setSimValueState(<SimValueState>)

        simXChange.setSimValueState(prey);
    } }
```

The pushSimXChangeData explained:

- This method is meant to push calculated data to the provided SimXChange object.
- Dependent on the state of the object either the number of preys must be forced (on initialization), or provided through the rate of change of the number of preys.
- There are no auxiliary variables, so no active code in this section.
- Every variable to be pushed to SimXChange must be mentioned in this section !

The full source code can be found in the example files.

4.2 The SimController implementation

We will describe here the simplest form of a SimController (called SimControllerModelPreyPredStandard), one that starts the SimPred and SimPrey objects, from the beginning of the simulation. If there is no SimController stopping these objects, they will automatically be stopped by the TimeDriver at the simulation's end date. Remember that a model can have more than one controller, each carrying out a specific task.

Pseudocode of testForSimObjectsToStart to start SimPredator and SimPrey once:

```
    public void testForSimObjectsToStart(final ArrayList<SimObject> simObjectsRunning) {

        if (!simXChange.isSimObjectClassNameRunning("SimPredator")) {
            // a SimPredator object is not yet running, start it
```

```
            String simID = SimPredator.class.getSimpleName();
            SimObject simObject = new SimPredator(simID, parXChange, simXChange);

            simObjectsRunning.add(simObject);
        }

        if (!simXChange.isSimObjectClassNameRunning("SimPrey")) {
            // a SimPrey object is not yet running, start it

            String simID = SimPrey.class.getSimpleName();
            SimObject simObject = new SimPrey(simID, parXChange, simXChange);

            simObjectsRunning.add(simObject);
        }
    }
```

The testForSimObjectsToStart method explained:

- This method is called at every time step of TimeDriver. It must start required SimObjects, return the count, through the return value, and a list of instantiated SimObjects, through the argument.
- The function simXChange.isSimObjectClassNameRunning returns a boolean whether there is a SimObject with the provided class name. If not, a SimObject of that class name must be started (in this case objects of SimPredator and SimPrey are instantiated).

4.3 The Model implementation

In general, a Model's implementation is the definition of the model at its highest level because this is where the list of instantiated controllers is governed and thus how SimObjects will run and interact. Although a Model's implementation can hold a hard coded list of to be instantiated controllers, in the example code you'll find cases where the controllers to be started are provided to the Model's implementation as an array of enum values, obtained from the provided ParXChange object. The pseudocode below shows the constructor of a very basic Model's implementation (called ModelPreyPred1) to instantiate the above described controller:

```
public ModelPreyPred1(ParXChange aParXChange,
                      SimXChange aSimXChange) {
    super(aParXChange, aSimXChange);

    SimController simctrl = new SimControllerModelPreyPredStandard(this, parXChange,
simXChange);
    simControllersRunning.add(simctrl);
}
```

The constructor method explained:

- This constructor is called only once for a model run.
- It instantiates the required controller(s) and add these objects to the list of running controllers. In larger models, probably more than one controller will be used (thus instantiated).

4.4 Setting up the model and running

An example method to run the ModelPreyPred1 model, starting at Jan 1, 2016, running for 100 days. Note that due to the large number of lines, labels (1), 2), 3) etc.) have been entered as comments and

are explained below. Also note that all parameterization is passed to the parXChange object through ParValue objects:

```
static void singlePreyPredRun() {
// 1)
    ParValue<LocalDate> startDate = new ParValue<>(LocalDate.of(2016,  1,  1) ,
                                                   ScientificUnit.NA);
    ParValue<LocalDate> endDate   = new ParValue<>(startDate.v().plusDays(100),
                                                   ScientificUnit.NA);
// 2)
    ParValue<Double> A = new ParValue<>(  0.01, ScientificUnit.NA, "A");
    ParValue<Double> B = new ParValue<>(  0.02, ScientificUnit.NA, "B");
    ParValue<Double> C = new ParValue<>(  10.0, ScientificUnit.NA, "C");
    ParValue<Double> D = new ParValue<>(   0.1, ScientificUnit.NA, "D");
    ParValue<Double> K = new ParValue<>(1000.0, ScientificUnit.NA, "K");
    ParValue<Double> R = new ParValue<>(   0.5, ScientificUnit.NA, "R");
// 3)
    ParValue<Double> PREDSTATE = new ParValue<>(   8.0, ScientificUnit.CNT_HA,
                                                "PREDSTATE");
    ParValue<Double> PREYSTATE = new ParValue<>( 200.0, ScientificUnit.CNT_HA,
                                                "PREYSTATE");
// 4)
    ParXChange parXChange = new ParXChange();

    parXChange.set(startDate);
    parXChange.set(endDate);

    parXChange.set(A);
    parXChange.set(B);
    parXChange.set(C);
    parXChange.set(D);
    parXChange.set(K);
    parXChange.set(R);

    parXChange.set(PREDSTATE);
    parXChange.set(PREYSTATE);
// 5)
    SimXChange     simXChange  = new SimXChange(runID);
    ModelPreyPred1 model       = new ModelPreyPred1(parXChange, simXChange);
    TimeDriver     timeDriver  = new TimeDriver(modelPreyPred1);
// 6)
    timeDriver.run(); // really run the model
// 7)
    int            stateID;
    ScientificUnit reqUnit;
    String         s;
    double         lastValue;

    stateID = simXChange.getTokenRead("PREDSTATE");
    reqUnit = ScientificUnit.CNT_HA;

    lastValue = simXChange.getValueAggregatedY(stateID,
                                               reqUnit,
```

```
                                                SimXChange.GroupY.LAST

    s = String.format("%s=%g (%s)", stateVarName, lastValue), reqUnit.toString());
    System.out.println(s);

    stateID = simXChange.getTokenRead("PREYSTATE");
    reqUnit = ScientificUnit.CNT_HA;

    lastValue = simXChange.getValueAggregatedY(stateID,
                                                reqUnit,
                                                SimXChange.GroupY.LAST

    s = String.format("%s=%g (%s)", stateVarName, lastValue), reqUnit.toString());
    System.out.println(s);
}
```

The singlePreyPredRun method explained:

1) Definition of start and end date. Create ParValue objects (startDate, endDate) which can hold LocalDate objects, and initialize with Jan 1, 2016, and 100 days later than Jan 1, 2016.
2) Definition of prey / predator parameterizations. Create ParValue objects which can hold double values for prey / predator parameterization, and initialize with often used default values.
3) Definition of initial states for predator and prey numbers (8 per ha for predators, 200 per ha for preys). Create ParValue object which can hold double values.
4) Declare the parXChange object and provide startDate, endDate, parameters and initial state values.
5) Instantiate two remaining system object and model specific object.
6) Run the simulation (filling the simXChange object).
7) Get final prey and predator number in number per ha and print to console.

4.5 Examining model output

After running the model (with `timeDriver.run()`), usually some kind of output processing has to take place, as can be seen in the previous chapter.

For a reference documentation of the SimXChange object, please refer to chapter 5.2.
Some examples are given below.

4.5.1 Output in Excel compatible file

An Excel compatible file of the data gathered in simXChange can be created with:

```
simXChange.report(<file name>);
```

This will create a comma separated values (.csv) file containing all data in simXChange (including a report section on all forcings).

4.5.2 Last value of a state variable

To obtain the last value of a state variable, one has to obtain a read-only token to the state first, after which that token can be used to query the simXChange object. In the prey / predator model, we get a token to PREDSTATE (the number of predators) by:

```
int stateID = simXChange.getTokenReadByVarName("PREDSTATE");
```

The last simulated value (is not necessarily the value on the last date of simulation !!) of PREDSTATE in number of animals per hectare can now be obtained by:

```
double d = simXChange.getValueBySimIDVarNameAgg(stateID,
                                                ScientificUnit.CNT_HA,
                                                SimXChange.GroupY.LAST);
```

Even if simulation continued for some reason without new values of PREDSTATE, we still would get the last value in the simXChange object.

There is a variety of grouping functions available which are all discussed in chapter 5.9.

4.6 Multiple model runs

Doing multiple model runs (allowing sensitivity analysis and calibrations) basically consists of the following steps:

- Make changes to the input parameters and / or initial states (in parXChange), example (give only parameter A a new value):
  `A.SetV(<new value>);`
  `parXChange.set(A);`
- Run the model:
  `timeDriver.run();`
- Process the output by querying the simXChange object. This can be simple to very detailed dependent on the purpose of the study. Sensitivity and continuity analysis, calibrations, regional and global model calculations etc..

5 WISS reference

5.1 ParXChange: Exchange of parameters and initial values directly

The ParXChange class is meant to handle all static exchange of data between the main program (where parameters and initial values are set), and the running SimObjects.

There are basically two ways to place static data in a ParXChange object, directly with the set method, or by using an intermediate ParValue object. An example of the first method is shown below. In this example a float parameter (with type double) is set and retrieved using the first method. Note that normally the retrieve is done somewhere else, probably in a running SimObject.

```
// 1)
ParXChange parXChange = new ParXChange();
parXChange.set("PAR_A", Double.class, true, 5.0, ScientificUnit.HA_KG);
// 2)
double par_A = parXChange.get("PAR_A", "documentationExamples",
                              Double.class, ScientificUnit.HA_KG);
System.out.println(String.format("PAR_A=%g", par_A));
```

1) Create a new instance of ParXChange, called parXChange and set the parameter PAR_A with value to 5.0, with unit hectares per kg as immutable (the "true" argument).
2) Declare and retrieve the double value of PAR_A with unit hectares per kg. The name of the caller here is "documentationExamples" (to make errors more readable). Print the name and value to the console.

The second method is used in the example below:

```
// 1)
final ParValue<Double> PAR_A = new ParValue<>(5.0, ScientificUnit.HA_KG, "PAR_A");

ParXChange parXChange = new ParXChange();
parXChange.set(PAR_A);
// 2)
double par_A = parXChange.get("PAR_A", "documentationExamples",
                              Double.class, ScientificUnit.HA_KG);
System.out.println(String.format("PAR_A=%g", par_A));
```

1) Identical to 1) from the first method, but now info is first given to a ParValue instance parValue (accepting only double data), before giving the parValue object to the parXChange object. Note that this set method gives immutable=false entries, enabling overwrites in a later phase.
2) Identical to 2) from the first example.

In the method below, an interpolation table is setup, defining the relation for variable XY_A where the X is in days, and the Y is in cm.

```
ParXChange parXChange = new ParXChange();
Interpolator XY_A = new Interpolator("XY_A", ScientificUnit.DAYS, ScientificUnit.CM);
XY_A.add( 0.0,  0.0);
XY_A.add( 6.0,  0.0);
XY_A.add(35.0, 24.0);
parXChange.set("XY_A", Interpolator.class, false, XY_A, ScientificUnit.NA);
```

The interpolation table can be obtained from parXChange in a manner similar to the earlier examples. For more details on interpolation, see the section on the Interpolator.

Note that PAR_A (or XY_A) is to be unique, but can be set more than once with the first method if the immutable flag is set to false. Otherwise calibration and other applications in which multiple runs have to be made with slightly changed parameter sets cannot be done. This also means that in order to get the variable, one has to know its name.

Procedures are similar for integer, text (strings) and basically any type of object (as long as they are Serializable).

5.2 SimXChange: How to register a state variable

To register a state variable with SimXChange, you must instantiate an intermediate object of type SimValueState. The example line below shows a state registration with a bounds check from a list of ranges.

```
private final SimValueState s = new SimValueState(arg1, arg2, arg3, arg4);
```

It is advised to declare s with `private final` because this protects against accidental changes.

- Arg1: the owner name as text of the state variable, the most logical choice being simID.
- Arg2: the name as text of the state variable.
- Arg3: must be one of the units of ScientificUnit (see section xxx for the current list).
- Arg4: the range type of the state variable, choose for example between ALL (all values allowed), ZEROPOSITIVE (only zero and positive values) etc.. For a complete list of predefined range types see the section on RangeUtils.

There is an overloaded constructor in which arg4 is split into 2 arguments for the allowed lower and upper bounds (as double datatypes). So this version has 5 arguments !

Registration of s with SimXChange takes place automatically on the initializing call to pushSimXChangeData() (must be mentioned there though, s.v must have received a value prior):

```
simXChange.forceSimValueState(s);
```

5.3 SimXChange: How to register an auxiliary variable

To register an auxiliary variable with SimXChange, you must instantiate an intermediate object of type SimValueAux. The example line below shows a registration with a bounds check from a list of ranges.

```
private final SimValueAux a = new SimValueAux(arg1, arg2, arg3, arg4);
```

It is advised to declare a with `private final` because this protects against accidental changes.

- Arg1: the owner name as text of the state variable, the most logical choice being simID.
- Arg2: the name as text of the state variable.
- Arg3: must be one of the units of ScientificUnit (see section xxx for the current list).
- Arg4: the range type of the state variable, choose for example between ALL (all values allowed), ZEROPOSITIVE (only zero and positive values) etc.. For a complete list of predefined range types see the section on RangeUtils.

There is an overloaded constructor in which arg4 is split into 2 arguments for the allowed lower and upper bounds (as double datatypes). So this version has 5 arguments !

Registration of a with SimXChange takes place automatically on the first call to pushSimXChangeData (must be mentioned there though, a.v must have received a value prior):

```
simXChange.setSimValueAux(a);
```

5.4 SimXChange: Setting rates of change

Setting rates of change can be done easily with the intermediate object which was required anyway to introduce a state variable. Setting the rate of change of the intermediate SimValueState object s, is simply:

```
s.r = <some value>;
```

In the pushSimXChangeData() method, the value is actually provided to SimXChange with:

```
simXChange.setSimValueState(s);
```

5.5 SimXChange: Get owned state variables

A SimObject can retrieve its own state variables prior to rate calculation in the pullSimXChangeData() method using the code (for intermediate SimValueState s):

```
simXChange.getSimValueState(s);
```

The .v property of s contains the value for the state. SimXChanges checks whether a valid value is available for the state at the current simulation date. An exception will be thrown otherwise.

5.6 SimXChange: Getting external variables (not knowing which SimObject published it)

Remember that in WISS, there is no difference between getting a state of another SimObject and getting an auxiliary variable from another SimObject. It can simply be seen as getting a time dependent external variable, or shorter, an external variable.

To obtain an external variable from SimXChange, it is best to instantiate an intermediate object of type SimValueExternal. The example line below shows the declaration and instantiation of such an intermediate object.

```
private final SimValueExternal e = new SimValueExternal(arg1, arg2, arg3);
```

It is advised to declare e with `private final` because this protects against accidental changes.

- Arg1: the name as text of the external variable.
- Arg2: must be one of the units of ScientificUnit (see section xxx for the current list).
- Arg3: the caller as text (usually the simID, for more readable error message).

N.B. remember that SimXChange does range checks for everything that it **receives**. In this particular case a value is **delivered**, so the range is already validated. Only the unit of delivery can be specified. Actually getting the value is done in pullSimXChangeData() with the line:

```
simXChange.getSimValueExternalByVarName(e);
```

SimXChange will look for a SimObject that published a proper value on the current simulation date. So it can be that on date x, the value obtained comes from SimObjectA, while on date y the values comes from SimObjectB. If no SimObject can be found on the current simulation date, a missing value (NaN) is

returned by SimXChange. The intermediate object e has two special boolean functions isMissing and isNotMissing which can be used to detect absent values for the external variable.

While the previous function of SimXChange tries to locate a value on the current simulation date, two other functions exist in which values can be obtained from another date. For a number of days relative to the current simulation date:

```
simXChange.getSimValueExternalByVarNameDelta(e, aDelta);
```

N.B. aDelta can only be negative, as there are no data at a later date then the current simulation date.

An external value on a specific date can be obtained through:

```
simXChange.getSimValueExternalByVarNameDate(e, aDate);
```

Where aDate must lie between the start date and the current simulation date (inclusive).

The above described way to obtain a value of an external variable is very well suited (and advised) for SimObjects. An alternative way to obtain such a value is by using the getValueBySimIDVarname method. This method returns a double value, but needs a token pointing to the requested simID and variable name. To let SimXChange look for a SimObject that published the variable "DVS" on the current date of simulation and return a readonly token (pointing to the simID of the publisher and variable name):

```
int tDVS = simXChange.getTokenReadByVarName("DVS", false);
```

The boolean false indicates that we do not want an exception if there is no valid value on the current date of simulation. If the token is valid, we can continue and obtain the actual value, using the token:

```
if (simXChange.isValidToken(tDVS)) {
    double DVS = simXChange.getValueBySimIDVarname(tDVS, ScientificUnit.NODIM);
further processing of DVS.........
}
```

In the examples above getTokenReadByVarName and getValueBySimIDVarname worked on the current date of simulation. However, there are also functions available that work on a relative and absolute date. Getting a token for DVS for the day before current simulation date is:

```
int tDVS = simXChange.getTokenReadByVarNameDelta("DVS", aDelta, false);
```

Processing the values goes with:

```
if (simXChange.isValidToken(tDVS)) {
    double DVS = simXChange.getValueBySimIDVarnameDelta(tDVS, aDelta,
ScientificUnit.NODIM);
further processing of DVS.........
}
```

Getting a token for DVS for a particular simulation date is:

```
int tDVS = simXChange.getTokenReadByVarNameDate("DVS", aDate, false);
```

Processing the values goes with:

```
if (simXChange.isValidToken(tDVS)) {
    double DVS = simXChange.getValueBySimIDVarnameDate(tDVS, aDate,
ScientificUnit.NODIM);
further processing of DVS.........
}
```

5.7 SimXChange: Forcing a state value

It is possible in SimXChange to override the value of any state in SimXChange. This is particularly necessary in the intervene() method of a SimObject which is meant to contain the state overrides should there be any. Forcing can be done on any intermediate object of type SimValueState. First we set the new value in the intermediate object after which it is communicated to SimXChange:

```
s.v = 10.0;
simXChange.forceSimValueState(s);
```

Please note that these forcings are logged internally in SimXChange (date of forcing, old value, new value) and will appear in the generated report file if the report method is called.

5.8 SimXChange: Getting an interpolator after simulation

It can be necessary in post processing to obtain the results for a variable by means of an interpolator object. Consider the crop development stage (DVS), for which we want to know on which date flowering was reached (DVS=1). This can be done by requesting an interpolator for DVS from SimXChange, with reversed X and Y, so that we can interpolate on DVS=1, and get the day since start of simulation on which this occurred. In code it goes (starts by getting a token):

```
int          tDVS        = simXChange.getTokenReadByVarName("DVS");
Interpolator dvs         = simXChange.getInterpolatorBySimIDVarName(tDVS, true);
Int          daysElapsed = MathUtils.roundToInt(dvs.interpolate(1.0));
```

5.9 SimXChange: Getting aggregated values after simulation

SimXChange facilitates running the simulation and producing an output report which is meant for interactive purposes where you can load the results into a spreadsheet, create graphs and further examine the data.

In more automated environments, you want to query SimXChange for certain aggregated values to be used in some other kind of processing. SimXChange facilitates that kind of use in various ways.

Suppose we want to find the last value of variable "WSO" in kg.ha-1 from SimObjectTest with simID="SimObjectTest". First we must obtain a readonly token for this:

```
int tWSO = simXChange.getTokenReadBySimIDVarName("SimObjectTest","WSO");
```

This gives us a 'handle (tWSO)' to get all sorts of aggregations for X, published by SimObjectTest. To get the last value:

```
double lastX = simXChange. getValueBySimIDVarNameAgg(tWSO, ScientificUnit.KG_HA,
SimXChange.AggregationY.LAST);
```

The following aggregations are available: FIRST=the first valid value 'ever', LAST = the last valid value 'ever', MIN = the lowest value 'ever', MAX = the highest value 'ever', COUNT = the count of values, SUM

= the sum of values, AVERAGE = the average of values, DELTA = the change in value from the first to the last (result = last - first) and RANGE = the highest value 'ever' minus the lowest value ever'.

Please note that these aggregations take missing values correctly into account by skipping them. However, the default for this function is to give an exception when no value at all could be found. If this is undesirable, you could call the overloaded version with an extra boolean argument indicating whether to check for not missing values.

In case a variable (e.g. WSO) was published by 2 or more SimObjects and we want to have the last value of all these SimObjects do the following:

```
ArrayList<Double> WSO = simXChange.getValuesByVarNameAgg("WSO", ScientificUnit.KG_HA,
SimXChange.AggregationY.LAST);
```

The values in WSO are given in SimID registration order.

SimXChange is also capable of aggregating over only a limited period of historic days, enabling the calculation of e.g. a moving average over the last 7 days. Suppose the token tTM points to the variable TM of a SimObject publishing it, the moving average over 7 days is (results in degrees Celsius):

```
double T7Days = simXChange.getValueBySimIDVarNameAggMoving(tTM, ScientificUnit.CELSIUS,
SimXChange.AggregationY.AVERAGE, 7);
```

Aggregation will take place over fewer days if we are at the beginning of simulation. On day 1, aggregation is over 1 day on day 2 over 2 days etcetera. Aggregation will remain on 7 days from day 7 onwards.

Inversely to getting the aggregated value for a variable (the min, max value and others), we may also want to know the simulation date on which this event occurred. Obviously a date cannot be found for the average, count, sum, delta and range aggregations, but finding a date is possible for the first, last, min and max values. For example getting the last date of WSO as published by SimObjectTest goes with:

```
LocalDate date = getDateBySimIDVarNameAgg(tWSO, SimXChange.AggregationDate.LAST)
```

Another kind of aggregation is to get a list of simulation dates where a variable crossed a given threshold value. If we want to know the date where WSO in kg.ha-1 exceeded zero (to get the date of start of yield formation) we could write (if there is only one SimObject publishing WSO, the array will contain only one element):

```
ArrayList<LocalDate> dates = simXChange.getDatesBySimIDVarNameCrosses(tWSO,
ScientificUnit.KG_HA, 0.0, true);
```

The last argument indicates whether we are interested in crossing from low to high values (true) or the other way around (false).

For the javadoc generated help, see section xxx.

5.10 SimXChange: Getting simulation dates in SimObjects

The SimXChange object can provide information about current date, elapsed number of days, etcetera. The following date info functions are available in the provided SimXChange object (for exact info see the javadoc reference xxx).

Function name	Meaning
getStartDate	Returns the start date of the simulation
getEndDate	Returns the end date of the simulation
getCurDate	Returns the current date of the simulation
elapsed	Returns the number of days that have passed since start of the simulation. On start date this is zero, on next day this is one, etcetera
maxDuration	Returns the maximum duration (in days) of the simulation, is essentially the number of days from start date till end date. Note that simulation will stop when there are no running SimObjects anymore (no sense continuing).
isOnStartDate	Returns flag whether the simulation is on the first date
isOnEndDate	Returns flag whether the simulation is on the end date
year	Returns the year of the current date of simulation.
month	Returns the month of the current date of simulation
dayInMonth	Returns the day in the month (1-31) of the current date of simulation
dayInYear	Returns the day in the year (1-365/366) of the current date of simulation

5.11 SimXChange: Iterating over all output variables

SimXChange contains the following functions to find out the contents of the object without knowing a priori what the contents will be (for exact info see the javadoc reference xxxx):

Function name	Meaning
getSimIDs	Returns a list of all registered SimIDs in order of registration.
getSimIDsByVarName	Returns a list of registered SimIDs that contain aVarName as variable name.
getSimIDsBySimObjectClassName	Returns a list of registered SimIDs of a particular SimObject class. Remember that if a particular SimObject is run only once, the SimID can be made equal to the classname, otherwise the SimID must be made unique in the instantiation in the SimController object for each time the SimObject is instantiated. For instance if a crop model SimCrop is run several times in one model run, the SimID for each instance could be as simple as suffixing the classname with a cultivation number.
getSimIDInfoBySimID	Returns an info object for the requested SimID. This info object contains the following properties: • simObjectClassName, the classname of the SimID, • startDateIndex, the number of days after start of simulation that the SimObject started, • endDateIndex, the number of days after start of simulation that the SimObject ended, • simIDState, the state of the SimObject (RUNNING, TERMINATED_NORMALLY or TERMINATED_ERROR) • simIDMsg, an optional text message given by the SimObject.
getElapsedBySimID	Returns the number of days that the SimID is active (e.g. from March 10 to March 11 is 1 day)
isSimObjectClassNameRunning	Info function that returns true if there is at least one reqistration with the mentioned ClassName.
getVarNameCount	Returns the number of registered variables.
getVarNameItem	Returns an info object for the requested index (valid values to be obtained from getVarNameCount. This info object contains the following interesting properties: • simIDVarName, the concatenated SimID and variable name,

	• simID, the SimID, • varName, the VariableName • simIDListIndex, index to list of SimIDss • isState, flag whether the item is a state, • scientificUnit, the unit of the variable, • lowerBound, the lower bound of the variable, • lowerBoundInclusive, flag whether the lower bound is inclusive, • upperBound, the upper bound of the variable, • upperBoundInclusive, flag whether the upper bound is inclusive, • varValues, array of values by dateIndex,
getStateVarForceCount	Returns the number of forced values, to be used in combination with getStateVarForceItemByIndex.
getStateVarForceItemByIndex	Returns an info object for the requested index (valid values to be obtained from getStateVarForceCount. This info object contains the following properties: • dateIndex, the number of days after start of simulation that forcing took place, • varListIndex, index to internal variable list. To be used with function getVarNameItem, • oldValue, the original value, • newValue, the forced value.

Below is some example code that outputs **all** variables in the provided simXChange object, using some of the methods described above:

```
int varNameCount = simXChange.getVarNameCount();
for (int i = 0; i < varNameCount; i++) {
    SimIDVarNameListItem item      = simXChange.getVarNameItem(i);
    int                  token     = simXChange.getTokenReadByVarName(item.varName);
    double               lastValue = simXChange.getValueBySimIDVarNameAgg(token,
item.scientificUnit, SimXChange.AggregationY.LAST);
    String               s         = String.format("%s.%s=%g (%s)", item.simID,
item.varName, lastValue, item.scientificUnit);
    System.out.println(s);
}
```

5.12 SimObject class

SimObject objects are the workhorse for doing simulation calculations. The minimal SimObject implementation is one in which only one auxiliary variable, or only one state variable is simulated. The constructor method needs to be programmed and intervene, auxCalculations, rateCalculations methods depending on the nature of the variables.

It is advised not to maintain states and auxiliary variables local to the SimObject instance for reason that these cannot be intervened from other locations, and that they are not queryable in the SimXChange object. It would imply a violation of the WISS principle. It is advised, however, to maintain parameters as local to the SimObject instance.

5.12.1 SimObject constructor method

When a SimObject constructor is called, the simID is received as it is provided by the SimController implementation creating the SimObject instance. It is the SimController's responsibility to provide a string that uniquely identifies this instance. Also the current ParXChange and SimXChange objects are provided, and via a call to the super method, accessible through the parXChange and simXChange names in every method of SimObject. The simID is used in communication with the provided SimXChange object.

The constructor method is also responsible for providing a majorversion, minorversion and description to the super object, so that an instantiator of the SimObject (a SimController) can verify that the version of the SimObject meets the minimum version requirement (with the checkMinimalVersion method).

When the SimObject implementation has state variables, the constructor is the logical place to initialize these.

How to obtain a parameter from parXChange
To obtain a float (double) parameter named p for use in a SimObject (see also the SimPrey code example for this):

```
private final double p = parXChange.get(arg1, arg2, arg3, arg4);
```

It is advised to declare p with `private final` because this protects p against accidental changes.

- Arg1: the name of the required variable. Now this could be ordinary strings, but the Model implementation seems to be the best place to centralize these.
- Arg2: the name of the caller (to be used in error message should these occur), best to use the provided simID for this.
- Arg3: the object type. For a double, fill in Double.class, for an integer, fill in Integer.class etc.
- Arg4: optional argument, must be one of the units of ScientificUnit (see section xxx for the current list). If Arg4 is left out, the returned value will be the native unit of the parameter.

How to obtain an interpolation table from parXChange
See the example below on how to obtain an interpolator from the provided parXChange object:

```
private final Interpolator xy = parXChange.get(arg1, arg2, Interpolator.class);
```

It is advised to declare xy with `private final` because this protects xy against accidental changes.

- Arg1: the name of the required interpolation table. Now this could be an ordinary string constant, but the Model implementation seems to be the best place to centralize these,
- Arg2: the name of the caller (to be used in error message should these occur), best to use the provided simID for this.
- Arg3: the object type, Interpolator.class in this case.

Note that the default for an object of type Interpolator is that extrapolation is not allowed (to safeguard simulation). See the section on the Interpolator to learn how to change this default behaviour.

How to register a state variable with simXChange

To register a state variable with the provided simXChange object it is best to instantiate an intermediate object of type SimValueState. The line below shows a state variable registration with bounds check from a list of ranges:

```
private final SimValueState s = new SimValueState(Arg1, Arg2, Arg3, Arg4);
```

- Arg1: the owner name as text of the state variable, the most logical choice being simID.
- Arg2: the name as text of the state variable.
- Arg3: must be one of the units of ScientificUnit.
- Arg4: the range type of the state variable, choose for example between ALL (all values allowed), ZEROPOSITIVE (only zero and positive values) etc.. For a complete list of predefined range types see the section on RangeUtils.

There is an overloaded constructor in which arg4 is split into 2 arguments for the allowed lower and upper bounds (as double datatypes). So this version has 5 arguments !

How to register an auxiliary variable with simXChange

To register an auxiliary variable with the provided simXChange object it is best to instantiate an intermediate object of type SimValueAux. The line below shows an auxiliary variable registration with bounds check from a list of ranges.

```
private final SimValueAux a = new SimValueAux(arg1, arg2, arg3, arg4);
```

It is advised to declare a with `private final` because this protects a against accidental changes.

- Arg1: the owner name as text of the state variable, the most logical choice being simID.
- Arg2: the name as text of the auxiliary variable.
- Arg3: must be one of the units of ScientificUnit.
- Arg4: the range type of the state variable, choose for example between ALL (all values allowed), ZEROPOSITIVE (only zero and positive values) etc.. For a complete list of predefined range types see the section on RangeUtils.

There is an overloaded constructor in which arg4 is split into 2 arguments for the allowed lower and upper bounds (as double datatypes). So this version has 5 arguments !

How to register an external variable with simXChange

To register an external variable with the provided simXChange object it is best to instantiate an intermediate object of type SimValueExternal. The line below shows an external variable registration. Note that there can be no range info given, the value is already checked by simXChange.

```
private final SimValueExternal e = new SimValueExternal(arg1, arg2, arg3);
```

- Arg1: the name of the required variable. Note that there is no owner required, the simXChange object will try to find the required variable even if it changed owners (there can be only one owner on a date of the system).
- Arg2: the unit in which the external variable is required.
- Arg3: the name of the caller, best to use the provided simID for this.

How to initialize a state variable at a zero or non-zero value

The first thing to do, of course, is to declare a state variable (see the section dealing with that). Suppose we have two state variables s1, and s2, of the preferred type SimValueState. We want the initial value of s1 to be zero (by design), and the initial value of s2 to be determined by the main program. To do this in code, see the example below (assuming SimExample as an example SimObject class):

```
public SimExample(String     aSimID,
                  ParXChange aParXChange,
```

```
                    SimXChange aSimXChange) {
    super(aSimID, aParXChange, aSimXChange, 1, 1, "a title", "a description");

    s1.v = 0.0;
    s2.v = parXChange.get(arg1, arg2, simID, Double.class, ScientificUnit.CNT_HA);

    this.pushSimXChangeData();
}
```

If this method is ended by a call to pushSimXChangeData, things will work out just fine. The meaning of the arguments of the .get() method are identical to the section "How to obtain a parameter from parXChange".

5.12.2 SimObject method: intervene()

The intervene() method is meant to be used to override one or more state variables in the provided SimXChange object. A second less frequent purpose could be to create or override parXChange data. Note, however, that any newly created entries in parXChange must be deleted on terminate of the SimObject. In the example code below, state variable s1 is halved (to emulate some harvesting etc.):

```
@Override
public void intervene() {
    super.intervene();

    this.pullSimXChangeData();

    s1.v = 0.5 * s1.v;
    simXChange.forceState(s1);
}
```

Contrary to other methods of SimObject this method should not be ended by a call to method pushSimXChangeData(), but the forceState() method of simXChange must be called right away. The simXChange object will store the value before and after forcing and the date of the forcing. This can be presented in a special type of report.

5.12.3 SimObject method: auxCalculations()

The auxCalculations() method is meant to calculate **only** auxiliary variables. In the example below, the auxiliary variable a is calculated as a linear interpolation in table XY on the external value of X:

```
@Override
public void auxCalculations() {
    super.auxCalculations();

    this.pullSimXChangeData();

    a.v = XY.interpolate(X.v(), ScientificUnit.NODIM, ScientificUnit.NODIM);

    this.pushSimXChangeData();
}
```

Note that any auxiliary variables must be declared as type SimValueAux in the field declaration section. Simply setting the .v (value) property to the new value is sufficient. The pullSimXChangeData() method

updates all local variables, the pushSimXChangeData() updates all local variables to the provided SimXChange object.

5.12.4 SimObject method: rateCalculations()

The rateCalculations() method is meant to calculate **only** rates of change of all variable that are declared as type SimValueState. In the example below, the rate of change (s.r) of state variable s is set as the state's value (s.v) multiplied by a relative growth rate rgr (usually gets a value in the object's constructor from some external parameter).

```
@Override
protected void rateCalculations() {
    super.rateCalculations();

    this.pullSimXChangeData();

    s.r = s.v * rgr;

    this.pushSimXChangeData();
}
```

The pullSimXChangeData() method updates all local variables, the pushSimXChangeData() updates all local variables to the provided SimXChange object.

5.12.5 SimObject method: canContinue()

This method is meant for the SimObject to request for termination to the simulation driver. A special case is when a SimObject must stop when **another** SimObject stops (a cascading effect of stopping one SimObject). Consider a case when the object to simulate a predator (SimPredator), must stop when the model for preys (SimPrey) stops (and vice versa). You could argue that the SimController that started both SimObjects could also know to stop both SimObjects. However, this may be true in many situations where there is only one SimController involved, but there is no guarantee, by design, that there is only one SimController, for WISS puts no limit on the number of SimControllers running simultaneously. For the SimPredator to know whether the SimPrey model has stopped, it would need to look at the interaction variable PREY that is used in SimPredator as external variable. Consider this example:

```
public boolean canContinue() {
    // request termination when the SimObject delivering prey no longer runs
    simXChange.getSimValueExternalByVarName(prey);

    return !prey.isTerminated();
}
```

when a SimObject needs to be terminated.

5.12.6 SimObject method: terminate()

This method is called by the simulation driver when the SimObject cannot continue (see previous section). The terminate() method is generally left empty. A case in which it is not empty is when the SimObject instance added entries to parXChange which need to be removed on destruction of the SimObject's instance. Another case could be a water balance calculation to verify that no water was lost during the calculations. In the example below

```
@Override
public void terminate() {
    parXChange.delete(arg1, arg2, arg3);

    super.terminate();
}
```

- Arg1: The owner of the parameter
- Arg2: The name of the parameter
- Arg3: The type of the parameter, usually Double.class, or Integer.class.

The terminate() method should never be called directly from within the object itself, instead it is called by the Model object.

Note there is no advantage here to call the pullSimXChangeData() and pushSimXChangeData() methods, since it is not very common that this method needs data from simXChange or is going to push data to simXChange.

5.12.7 SimObject service methods: pullSimXChangeData() and pushSimXChangeData()

These methods do not have an essential role in the simulation, but are there to obtain more readable code. Please follow the code conventions of these methods. The example below shows a simple pullSimXChangeData () example in which the external variable e1 is obtained and the new value for the owned state s1.

```
private void pullSimXChangeData() {

    // get external variables
    simXChange.tryValue(e1);

    if (this.isAuxCalculating() ||
        this.isRateCalculating()) {
        // get owned states
        simXChange.getSimValueState(s1);
    }
}
```

Note that is not necessary to obtain owned auxiliary variables ! The tryValue() method should be used to obtain external variables as this method will try to locate another simObject if the required variable is not published anymore by the previous simObject. This is a very essential functionality! A NaN value is returned if the variable is no longer published by any of the simObjects. The boolean functions isMissing() or isNotMissing() can be used to find out what the status is of the variable is after tryValue().

The example below shows a simple pushSimXChangeData() example in which the SimObject contains a state variable (s1) and an auxiliary variable (a1).

```
private void pushSimXChangeData() {

    if (this.isInitializing()) {
        simXChange.forceState(s1);
    } else if (this.isAuxCalculating()) {
        simXChange.setAux(a1);
```

```
    } else if (this.isRateCalculating()) {
        simXChange.setRate(s1);
    }
}
```

All owned state variables must occur in the isInitializing section and in the isRateCalculating section as a general rule. Similary the owned auxiliary variables must occur in the isAuxCalculating section.

5.12.8 Working with simulation date in a SimObject

It is sometimes necessary to know the date of simulation or the number of elapsed days etc.. In general these can be obtained from the provided simXChange object. The example below shows how to trigger on the number of elapsed days:

```
if (simXChange.elapsed() = 10) {
    // do something
}
```

Or how to use the simulation date for something (on Jan 1, 2010):

```
LocalDate curDate  = simXChange.getCurDate();
if (curDate.equals(LocalDate.of(2010, 1, 1))) {
    // do something
}
```

5.13 SimController class

SimControllers are the objects that instantiate (=start) and stop (=terminate) SimObjects. For starting one or more SimObjects the method testForSimObjectsToStart needs to be properly implemented, for stopping one or more SimObjects the testForSimObjectsToTerminate needs to be properly implemented.

The specific SimControllers that are required to run the model need to be instantiated before the start of simulation and will continue to run until the end. The implementations for testForSimObjectsToStart and testForSimObjectsToTerminate need to be programmed to properly handle this. Some examples for starting and stopping will be given in the next sections.

Please note that in testForSimObjectsToStart, SimObjects are actually instantiated, while in testForSimObjectsToTerminate, only a list of SimObjects to be terminated is returned, the actual termination taking place somewhere else.

5.13.1 SimController method: testForSimObjectsToStart

Important to realise is that this method must add the started SimObject(s) to the list provided through the argument list (ArrayList<SimObject> simObjectsRunning)

In a SimController which deals only with terminating SimObjects, this method must have the following dummy implementation (returning zero started SimObjects):

```
public void testForSimObjectsToStart(ArrayList<SimObject> simObjectsRunning) {
}
```

A simple case is when a SimObject must be started if none of its class are running. This comes down to starting the SimObject at the beginning and restarting one whenever a previous one got terminated:

```
if (!simXChange.isSimObjectClassNameRunning(SimPredator.class.getSimpleName()))
    simIDSimPredator = SimPredator.class.getSimpleName();
    SimObject simObject = new SimPredator(simIDSimPredator, parXChange, simXChange);

    simObjectsRunning.add(simObject);
}
```

Please note that in the last 2 lines, the instantiated SimObject is added at the end of the list. Please note also that code to log the instantiation has been removed here for clarity but you can find this in the examples.

In some cases it is required to not add the instantiated SimObject at the end of the list, but somewhere in between. As an example let's assume that we want to add the instantiated SumCropPhenoType1 object after the SimObject that published ET0. This can be programmed by:

```
final int    token = simXChange.getTokenReadByVarName("ET0", true);
final String simID = simXChange.getSimIDFromToken(token);
int          i     = model.GetSimObjectsRunningIndexBySimID(simID, true);

i++;
simObjectsRunning.add(i, simCropPhenoType1);
```

First we obtain a token of the variable of the SimObject that publishes ET0, then we get the SimID of this object. We subsequently ask the Model for the position in the list of running SimObjects where this SimID is put, and then insert it after that location.

5.13.2 SimController method: testForSimObjectsToTerminate

This method must return a list with SimObjects that need to be terminated but not actually terminating them. Each item in the list is, on return of the function, terminated (starting at the beginning of the list) by the Model.
There are basically 2 methods by which a SimObject can be stopped before reaching the end simulation date. These are:

1. The SimObject itself requests termination through the canContinue() function,
2. A running SimController decides to stop the SimObject.

We do not have to deal in the SimController with the termination of a SimObject through the canContinue() function. This type of termination is dealt with by the Model and takes place **after** the SimController's triggered terminations. The reason for this is that, through the SimController's triggered termination, one or more SimObjects cannot continue because the necessary external variables are not there anymore. Examples of SimObject's canContinue implementations can be found in xxx.

An example of a SimController's termination is given below, in which the SimObject publishing the PRED state is terminated when that state's value reaches 15:

```
public ArrayList<SimObject> testForSimObjectsToTerminate() {

    final ArrayList<SimObject> result = new ArrayList<>();

    final int tPRED = simXChange.getTokenReadByVarName("PREDSTATE", false);
    if (simXChange.isValidToken(tPRED)) {
```

```
        final double PRED = simXChange.getValueBySimIDVarname(tPRED,
ScientificUnit.CNT_HA);

        if (PRED > 15.0) {
            final String    simID     = simXChange.getSimIDFromToken(tPRED);
            final SimObject simObject = this.terminateAndGetSimObjectBySimID(simID,
false, "threshold reached");
            if (simObject != null) {
                result.add(simObject);
            }
        }
    }
    return result;
}
```

First we instantiate the list to return. We try to get a read-only token for PREDSTATE, but anticipate that it can be missing. If it is not missing we can safely obtain the value in number per hectare. When that number exceeds 15, we are going to terminate the SimObject publishing it. We get the SimID of the SimObject publishing it, and call the protected SimController's function to obtain the object from the SimID, giving it a termination reason "threshold reached". The actual termination method of the SimObject will be called inside the terminateAndGetSimObjectBySimID function.

5.14 Model class

The Model subclass implementation is where everything is integrated. Primary responsibilities are:

1. To keep a list of **all** model specific names (the names of the states, auxiliary variables, parameters etc.), as static constants,
2. To instantiate the right SimControllers to do something sensible in the whole model. The list of instantiated SimControllers is kept in the local object simControllersRunning. There are no special methods managing the list of running SimControllers since they always run from the first date of simulation until the last day of simulation.

An example list of static names could be (for the predator / prey model):

```
// for SimPrey ------------------------------------------------------------
public static final String PREYSTATE  = "PREYSTATE";

// for SimPredator --------------------------------------------------------
public static final String PREDSTATE  = "PREDSTATE";

// Parameter names for initialization
public static final String A = "A";
public static final String B = "B";
public static final String C = "C";
public static final String D = "D";
public static final String K = "K";
public static final String R = "R";
```

The instantiation of SimControllers should take place in the constructor of the Model object. Here is an example for ModelPreyPred1 (some unimportant code removed):

```
public ModelPreyPred (ParXChange aParXChange,
                      SimXChange aSimXChange) {
```

```
    super(aParXChange, aSimXChange);

    title        = "ModelPreyPred1";
    description  = "Calculation of prey / predator system";

    simctrl = new SimControllerModelPreyPredStandard(this, parXChange, simXChange);
    simControllersRunning.add(simctrl);
}
```

5.15 Running the model

Below are the basic calls to make a run of ModelPreyPred1 in WISS and get output in ModelPreyPred1.csv:

```
// creation of parXChange, setting dates, parameters and initial state values
parXChange = new ParXChange();
parXChange.set(startDate);
parXChange.set(endDate);
parXChange.set(A);
parXChange.set(B);
parXChange.set(C);
parXChange.set(D);
parXChange.set(K);
parXChange.set(R);
parXChange.set(PREDSTATE);
parXChange.set(PREYSTATE);

simXChange = new SimXChange(runID);
model      = new ModelPreyPred1(parXChange, simXChange);
timeDriver = new TimeDriver(model);

timeDriver.run(); // really run the model

simXChange.report("ModelPreyPred1.csv"); // export model output to a .csv file
```

In this piece of code, basically 4 objects need to be instantiated: a ParXChange object for holding model input, a SimXChange object for holding model output, a ModelPreyPred1 object and a TimeDriver object (receiving the model object as the object to run).

Please note that startDate, endDate, A, B, C, D, K, R, PREDSTATE and PREYSTATE are actually ParValue objects that facilitate setting values in parXChange. These objects were instantiated and parameterized earlier in the code but were left out here for clarity (for a discussion on the ParValue class see section xxx).

The .report method of simXChange produces the .csv file ModelPreyPred1.csv, containing date of simulation and unit and values of all variables stored in the SimXChange object. Also state variable forcings are reported below the block of time series. The top part of the .csv file (the block of time series) of a run with ModelPreyPred1 is given below. The DATE and ELAPSED columns are standard columns of SimXChange, the other columns are model specific:

// Column units:	Days	[no.ha-1]	[no.ha-1]
DATE	ELAPSED	PREDSTATE	PREYSTATE

01-01-2016	0	8	200
02-01-2016	1	7.49003	265.498
03-01-2016	2	7.09033	345.538
04-01-2016	3	6.7956	437.894
05-01-2016	4	6.59799	536.867

6 Other standard WISS components

Here we will discuss some utility components of the WISS framework that a modeller may need.

6.1 Interpolator in nl.wur.wiss.mathutils

The interpolator class is meant for linear interpolation in a set of X, Y values.

An interpolator object can be instantiated in three ways:

1. by providing an array with X values and one with Y values (both need to be equally long)
2. by providing a 2-dimensional matrix with 2 columns of X and Y values,
3. by simple instantiation, not providing Y, Y data yet. Setting up X and Y values should be done with the .add method

Note: units for provided X and Y data are provided in all instantiations, also note that X values must be provided in **ascending** order to the Interpolator object, else a blocking error will occur.

The interpolator object basically finds a linearly interpolated Y value at a given X. If the given X value is outside the range of X values, the extrapolation setting determines what will happen. There are three extrapolation types:

1. extrapolation is not done, an error will occur, providing the ID of the interpolator and the X value, this is the default behaviour,
2. extrapolation is done by returning the Y value at the end at which the extrapolation occurs (constant extrapolation),
3. extrapolation is done by returning the Y value determined from the slope of the Y values at the end at which extrapolation occurs.

With the interpolate method, the value to interpolate from and the interpolation result are assumed in native units, however there is an overloaded method in which the X can be provided in another unit, and Y be delivered in another unit.

This class is optimized for performance through several techniques. Firstly the slopes of all segments are determined once so that the interpolated value calculation is more straightforward. Secondly, the last segment of interpolation is stored and tried at a new interpolation, increasing the likelihood that the right segment for interpolation is found immediately.

For further reference see the javadoc sections xxx.

6.2 MathUtils in nl.wur.wiss.mathutils

This package currently contains only one function: doubleToInt. This function is useful in cases where a double value needs to be rounded to an integer. Since the range of a double is larger than that of an integer, it is best to check against the valid range of an integer. This is exactly what this function does.

6.3 RangeUtils in nl.wur.wiss.mathutils

This package is meant to deliver functionality concerning ranges. It contains the following functions:

Name	Meaning
inRange	Returns a flag whether the first argument is numerically in the range of the second and third argument, bounds are inclusive. Versions exist for testing int, double and LocalDate types.

ensureRange	Ensures that the returned value is inside the bounds specified by the second or third argument. Versions exist for int and double types.
isMissing	Returns a flag whether the LocalDate argument is empty.
isNotMissing	The inverse of isMissing.
getLowerBound	Returns the lower bound (as a double) for the different range types.
getUpperBound	Returns the upper bound (as a double) for the different range types.
safeExpr	Meant to evaluate an expression argument. An exception is thrown if the expression results in a divide by zero, positive or negative infinity.

6.4 DateUtils in nl.wur.wiss.core

Provides utility functions for working with dates (as type LocalDate). It contains the following functions:

Name	**Meaning**
diffDays	Returns the number of days between 2 dates, if the dates are the same, the number is zero.
getLocalDate	Returns a single date when composed by a year and the day number in that year.

6.5 ParValue in nl.wur.wiss.core

ParValue is the generic parameter value class that can hold all types of data. Objects of this class can easily be 'given' to a ParXChange object. ParValue objects contain the name of the parameter, the value, etc. See some examples below:

For setting up a parameter A (used in the Prey / Predator model):

```
ParValue<Double> A = new ParValue<>(0.01, ScientificUnit.NA, "A");
```

The parameter object A can now be given to parXChange by:

```
parXChange.set(A);
```

This is equivalent to:

```
parXChange.set("A", Double.class, 0.01, ScientificUnit.NA);
```

6.6 ScientificUnitConversion in nl.wur.wiss.core

This class contains the unit conversion function(s) to convert values from one unit into another. Actual unit conversion is done in the private function _convert in that class. It could well be that the current implementation is incomplete because the number of possible unit conversion is seemingly endless. Please contact us if there are any essential conversions missing.

A unit conversion basically goes like this (to convert 20 degrees Celsius into degrees Fahrenheit):

```
double tempInC = 20.0;
```

```
double tempInF = ScientificUnitConversion.convert("tempInC", tempInC,
ScientificUnit.CELSIUS, ScientificUnit.FAHRENHEIT);
```

Note that the "tempInC" argument is there so conversions errors can be made more readable and facilitates location of the problem in code. Available units and unit conversions can be found in the Javadoc section (xxx).

6.7 SimValueState in nl.wur.wiss.core

Objects of SimValueState are meant to be used in SimObjects as placeholders for information pertaining to one state variable. This object will hold the token of the state, the current value, the rate (if set), the previous value (if there is one), the scientific unit, the owner, the lower bound and the upper bound. It can be used to exchange information with SimXChange. Please follow the SimObject examples on how to instantiate, set the rate, use the integrated state value, etcetera.

Remember that in case other SimObjects are interested in the state value of a SimValueState object, they must be using objects of class SimValueExternal. See the WISS example files and JavaDoc on how exactly instantiation must take place.

A SimValueState object has the following properties (only most important ones shown):

Name	Meaning
v	Value of the state
r	Rate of change (is NaN after integration)
vp	The state value of the previous time step (Double.NaN on first day)

6.8 SimValueAux in nl.wur.wiss.core

Objects of SimValueAux are meant to be used in SimObjects as placeholders for auxiliary variables (variables that are time dependent but can be determined without rate integration, thus non-state variables). Since these variables are not integrated by the provided SimXChange object (but values are kept in SimXChange!), a previous values could not be guaranteed to be of the previous day. See the WISS example files and JavaDoc on how exactly instantiation must take place.

The property .v (=value) is the SimValueAux most important property.

6.9 SimValueExternal in nl.wur.wiss.core

Objects of SimValueExternal are meant to be used in SimObjects as placeholders to obtain state and auxiliary variables from other SimObjects. Remember that there is no reason to distinguish between the two types outside the SimObject where they are determined. See the WISS example files and JavaDoc on how exactly instantiation must take place.

A SimValueExternal object has the following properties (only most important ones shown):

Name	Meaning
v	Value of the state
terminated	Whether the variable is terminated on the current simulation day

7 Future of WISS

WISS with the implemented WOFOST model is currently used in a couple of projects that aim to do regional crop forecast under prevailing weather conditions. Due to the excellent performance of WOFOST in the WISS framework, there is currently no need to make the WISS framework more efficient. It could well be that such a need will arise with other models getting computation intensive WISS implementations. To be able to store the complete model state and restart from that state could be advantageous in cases where a crop model runs until the current day with observed weather, and the remainder of the season with generated weather.

8 Where to obtain and licensing

Hoe kan iemand de WISS source code bemachtigen?