# Reuse of process-based models: automatic transformation into many programming languages and simulation platforms

Cyrille Ahmed Midingoyi[1,2], Christophe Pradal[2,3]*, Ioannis N. Athanasiadis[4], Marcello Donatelli[5], Andreas Enders[6], Davide Fumagalli[7], Frédérick Garcia[13], Dean Holzworth[9], Gerrit Hoogenboom[10,12], Cheryl Porter[11], Hélène Raynal[8], Peter Thorburn[12], Pierre Martre[1,]*

[1] LEPSE, Univ Montpellier, INRAE, Institut Agro, Montpellier, France

[2] AGAP, Univ Montpellier, CIRAD, INRAE, Institut Agro, Montpellier, France

[3] LIRMM, Univ Montpellier, Inria, CNRS, Montpellier, France

[4]Wageningen University, Wageningen, The Netherlands

[5] Research Centre for Agriculture and Environment, CREA, Bologna, Italy

[6] Institute of Crop Science and Resource Conservation (INRES), University of Bonn, Bonn, Germany

[7] Institute for Environment and Sustainability, Joint Research Centre, European Commission, Ispra, Italy

[8]AGIR, INRAE, Castanet-Tolosan, France

[9] CSIRO Agriculture and Food, Toowoomba, Australia

[10] Institute for Sustainable Food Systems, University of Florida, Gainesville, USA

[11] Agricultural & Biological Engineering, University of Florida, Gainesville, USA

[12] CSIRO Agriculture and Food, Brisbane, Australia

[13]MIAT, INRAE, Castanet-Tolosan, France

***** Corresponding authors' e-mail christophe.pradal@cirad.fr; pierre.martre@inrae.fr

## Abstract

The diversity of plant and crop process-based modeling platforms in terms of implementation language, software design, and architectural constraints limits the reusability of the model components outside the platform in which they were originally developed, making model reuse a persistent issue. To facilitate the intercomparison and improvement of process-based models and the exchange of model components, several groups in the field joined to create the Agricultural Model Exchange Initiative (AMEI). AMEI proposes a centralized framework for exchanging and reusing model components. It provides a modular and declarative approach to describe the specification of unit models and their composition. A model algorithm is associated with each model specification, which implements its mathematical behavior. This paper focuses on the expression of the model algorithm independently of the platform specificities, and how the model algorithm can be seamlessly integrated into different platforms. We define CyML, a Cython-derived language with minimum specifications to implement model component algorithms. We also propose CyMLT, an extensible source-to-source transformation system that transforms CyML source code into different target languages such as Fortran, C#, C++, Java and Python, and into different programming paradigms. CyMLT is also able to generate model components to target modeling platforms such as DSSAT, BioMA, Record, SIMPLACE and OpenAlea. We demonstrate our reuse approach with a simple unit model and the capacity to extend CyMLT with other languages and platforms. The approach we present here will help to improve the reproducibility, exchange and reuse of process-based models.



**Keywords:** Source transformation, model reuse, transpiler, software reuse, crop model.

# 1. Introduction

Process-based crop models (PBM) are increasingly developed for a wide range of applications and research purposes. Even though there are key biophysical processes in PBM such as phenology, soil water balance, or biomass production, their modeling differs from one model to another according to the biological details, influenced by the availability of input data and final use of the model. The choice of modeling approaches to represent processes and combine them is also one of the main reasons which led to the development of multiple PBM to simulate the same crops (Jones et *al.* 2017). They have often been written repeatedly in several different languages with different software architectures. For example, the WOFOST model is implemented in Fortran in the WOFOST Control Centre (WCC) package, in Python in the Python Crop Simulation Environment framework, in Java in the Wageningen Integrated Systems Simulator framework (WISS), in C# in the Biophysical Models Application (BioMA) framework, and in C++ in the Crop Growth Monitoring System (CGMS) (de Wit *et al.* 2019; van Kraalingen *et al.* 2020).

The diversity of PBM has motived the development of different initiatives that intend to compare their performance and improve them by integrating new scientific knowledge to target the next generation of crop models (Rosenzweig *et al.* 2013; Bindi *et al.* 2015). PBM intercomparison studies (Palosuo et *al.* 2011; Rötter et *al.* 2011; Asseng et *al.* 2013; Aslam et *al.* 2017) have pointed out the variability in model outputs but often without quantifying the sources of uncertainty or analyzing the processes involved. These studies showed the potential and limits of PBM and highlighted the need to evaluate them at the process level, but also to exchange model parts (components) between models (Donatelli et *al.* 2014; Muller and Martre 2019). PBM are increasingly implemented as autonomous components describing each biophysical process. However, there is currently little exchange and reuse of PBM components between modeling groups despite theoretical and application interests (Holzworth et *al.* 2014). The main limitation comes from compatibility issues between PBM platforms (frameworks) resulting from differences in programming languages that are used and their specificities.

The modeling frameworks used in agricultural modelling depend on the programming language in which they have been implemented, the software design, and code conventions they use. For example, the crop modeling frameworks APSIM Next Generation (Holzworth et *al.* 2018) and BioMA (Donatelli *et al.* 2010) are based on component-oriented techniques and require models to be developed in C#. DSSAT (Jones et *al.* 2003; Hoogenboom et *al.* 2019) and STICS (Brisson *et al.* 1998) provide generic crop modules in Fortran with a procedural approach that can be specialized for different species. Simplace (Enders *et al.* 2010) uses the Java language, while Record (Bergez *et al.* 2016) uses C++; both require that their components share a built-in interface. Therefore, model components can be reused in a given platform but their reuse in other platforms remains difficult. Existing solutions that couple models written in different languages are rather technical (generation of

wrappers) or low level (reading and writing in files). We propose here an abstraction, a sharing language, and a transformation system, based on the scientific content of the model, i.e., its algorithms. Multilanguage and integrated modeling frameworks like OpenAlea (Pradal *et al.* 2008, 2015) and yggdrasil (Lang 2019) offer a language binding approach to provide third-party developers with a choice of languages (Villa 2001; Lang 2019). Therefore, they overcome the difficulty of implementing algorithms efficiently in high-level languages. However, they do not provide a solution to the reuse or exchange of models between frameworks. In these platforms, models are reused as black boxes and the integrated models, therefore, lack the required transparency. Moreover, this approach requires knowledge of the frameworks they integrate and the deployment of the core of each framework. Domain-specific programming languages that are agnostic to a specific programming language have also been proposed as a solution to the problem (Athanasiadis and Villa 2013; Villa *et al.* 2017) aiming to support interoperability with rich semantics.

To facilitate PBM component exchange, several groups in the field have joined forces to create the Agricultural Model Exchange Initiative (AMEI; Martre et *al.* 2018). AMEI brings together some of the most widely used crop modelling and simulation platforms, including APSIM, BioMA, DSSAT, OpenAlea, RECORD, Simplace and other crop models such as STICS and *SiriusQuality* (Martre *et al.* 2006) The vision of AMEI is to (*i*) increase capabilities and responsiveness to model developers' needs; (*ii*) use modular modelling to share knowledge and rapidly develop operational tools; (*iii*) reuse model parts to leverage the expertise of third parties; (*iv*) renovate legacy code; and (*v*) realize the benefit of sharing and complementing different expertise.

Based on a declarative modeling approach (Athanasiadis *et al.* 2011), AMEI proposes a centralized framework (Crop2ML; Midingoyi et *al.* 2020) to exchange and reuse model components. Crop2ML provides a meta-language based on shared concepts between crop simulation platforms to describe specifications of model components and compositions. A model algorithm describes the behavior of the component in terms of the sequence of inputs, successive rules or actions, conditions or a flow of instructions from inputs to outputs including mathematical expressions. A model algorithm is associated with each model specification. After a modeler has represented the specifications of its model, two relevant questions remain to be answered: (1) How can a model algorithm be described independently of the platform specificities; and (2) How can it be seamlessly integrated into existing simulation platforms?

Similar approaches have been used in the Systems Biology community where several domain-specific modeling standard languages including SBML, CellML, and NeuroML have been designed to exchange and store models (Autumn Cuellar *et al.* 2006; Gleeson *et al.* 2010; Hucka *et al.* 2015). These XML-based languages provide specific elements to describe model structure and equations

using Mathematical Markup Language (MathML; Ausbrooks et *al.* 2003) that describes mathematical notations and captures both its structure and content. However, these languages are limited to specific formalisms (e.g. chemical reactions, differential equations) and cannot be easily extended to represent crop models in their full complexity and diversity. System Biology languages support model transformation from one standard to another (e.g. form CellML to SBML; Schilstra et *al.* 2006) and from XML to executable code. In contrast, Crop2ML provides models as components that can be integrated into simulation platforms. Therefore, our design choice was to introduce a general programming language to represent complex control flow such as loops or conditions statements.

In this paper, we present CyML, a Cython-derived language (Behnel *et al.*, 2011) with minimum meta-specifications to implement algorithms of Crop2ML models. This language allows encoding the model algorithm independently of any crop modeling platform and implementation language. We also propose CyMLT, a source-to-source transformation system. This one-to-many transpiler transforms CyML source code into different target languages such as Fortran, C#, C++, Java and Python. CyMLT is also able to directly generate components to target modeling platforms such as DSSAT, BioMA, Record, SIMPLACE and OpenAlea. Differences between platforms are not only due to the languages used to implement models but also to the software architectural design choices and modeling conventions. For instance, model components in PMF (APSIM next generation) and BioMA are written in C# in both platforms but the reuse of PMF components in BioMA (and vice versa) can only be done at the level of binaries, and, therefore, as black boxes. CyMLT takes into account platform requirements to generate model components that are compliant with existing platforms. Source to source transformation is a well-established solution used to address software reuse issues (Plaisted 2013; Fernique and Pradal 2017). It transforms source code from a high-level language to another one. However, to the best of our knowledge, no solution exists that targets PBM component reuse using automated source-to-source transformation. In this paper we present this issue by focusing on code reuse and reproducibility to enhance collaboration between crop modelers and to facilitate model coding for non-programmers, while keeping the transparency of model constructs.

Different source-to-source transformation systems are available for different purposes, both commercial (e.g. Baxter et *al.* 2004) and open source (Quinlan and Liao 2011). Some lessons can be learned from these approaches. Many source-to-source transformation systems take as input a subset of one language and transform it to a single target language with specific transformation purposes without showing their extensibility (Akeret *et al.* 2015; Bysiek et *al.* 2017; Misse-chanabier et *al.* 2019). Few one-to-many (Plaisted, 2013; Schaub and Malloy, 2016) and many-to-many (Baxter et *al.* 2004) solutions have been proposed. They usually define a subset of language features and are based on a common intermediate representation of the languages provided from their similarities. However, they do not consider transformation between different programming paradigms. For instance, to our

knowledge, there is no system that transpiles from a procedural algorithm to both a procedural and an object-oriented program. To avoid losing assumptions or domain knowledge such as code documentation or variable units, a PBM source-to source-transformation should also integrate domain specific knowledge to generate code that is easy to read, following developer guidelines specific to each language.

First, we present the design and implementation of CyML language and the one-to-many transformation workflow. Then we demonstrate the use of CyML and for a simple model component, which simulates wheat shoot number and the extensibility of CyMLT to new languages or simulation platforms. Finally, we discuss our results and present some perspectives. This paper is not intended to provide a full description of the language and its transformation but uses them to demonstrate that a model algorithm can be implemented once and be used to generate reusable and reproducible model components in different target languages and platforms.

# 2. Methods

## 2.1 Brief overview of Crop2ML

Crop2ML has been developed to offer to the crop modeling community a common framework for crop model component development, exchange, and reuse. It provides a model component specification language based on XML meta-language. It consists of unified concepts and elements allowing to describe a biophysical process regardless of the simulation platform. A Crop2ML model is an abstract model that may be either a unit model with fine granularity or a composite model represented as a graph of unit models connected by their inputs and outputs to manage model complexity. Crop2ML separates model specification from model algorithm. A model specification contains formal descriptions of the model, the inputs, outputs, state variable initializations, auxiliary functions and a set of parameters and unit tests. Thus, it allows for checking that a model reproduces the expected outputs values with a given precision. It supports multiple tests associated to one or multiple set of parameters' values. However, baseline parameter sweeps are not supported due to limited support in various languages and unit test frameworks. The specification also contains the algorithm written in CyML and any auxiliary functions called from the model algorithms or in other functions. They reduce code length and, therefore, improve readability of model algorithm by promoting reuse and increasing abstraction. Auxiliary functions include mathematical functions such as interpolation, and lower and upper bound functions.

All model units and composite models are then transformed into different languages or simulation platforms to be incorporated into modelling platforms.

The source code (https://github.com/AgriculturalModelExchangeInitiative/Crop2ML) and full documentation (https://crop2ml.readthedocs.io/en/latest/) of Crop2ML are available on Github.

## 2.2 Requirements and CyML design choices

We designed the CyML language to meet the following requirements.

**(*i*) *Keep compatibility with programming languages of crop simulation platforms.*** A model can be reused if it can be separated from its original platform and expressed using equivalent and explicit constructs available in all supported programming languages and platforms. Therefore, a sub-language needs to be identified that is minimal enough to express biophysical processes in all platforms but expressive enough to capture the complexity of most models. The resulting code must be removed from the technical subtleties of the platform but it will still depend on the platform language. In fact, most of these languages are direct descendants of the C language from which they inherit some constructs. Thus, they provide some similarities such as statements, the sequencing controlled by loop and conditional constructs, and functions that foster program modularization (Akin 2003). This leads to the ability to define a common language based on their common features. This language must be chosen in such a way that all its constructs are mapped to the constructs of the target languages, thus producing a fully automated source to source transformation. It must also provide some mathematical standard functions that have their equivalents in the language of the modeling platforms.

**(*ii*) *Link model specification and model algorithm to keep domain knowledge.*** As the model specification language is separated from the language of the algorithms in Crop2ML, it is necessary to provide and link domain knowledge information, including the context or decisions underlying the algorithm and its implementation in the language. It is also important to reduce the coding role of modelers in the implementation of model algorithms so that they can focus on the scientific knowledge (Brown et *al.* 2018). Our hypothesis is that model reuse can be achieved if its algorithm is closely associated with its specification. Thereby model specification can be used to generate a function signature or domain class from the description of inputs and outputs. The specification must also allow pass through documentation within the translated source code, but also to validate model algorithms with the unit tests they incorporate.

**(*iii*) *Cover the domain of interest.*** The abstract language must be sufficient to implement a biophysical process. This means that it must include all relevant and minimal features such as data types, modularity, and structures to encode any model algorithm. For example, in order to encode a model algorithm based on a set of mathematical expressions, a simple pseudo-code described as a

sequence of assignment statements are suggested. Like the model specification, this language must be modular. Model algorithms must be self-contained and reusable within a composite model.

**(*iv*) *Have a gentle learning curve.*** An important impact of the language is its learning curve, which must be shallow and allow modelers to focus on the science of the model rather than on its implementation. Thus, CyML must enable an optimal model developer experience with a learning curve that does not intimidate new users. The algorithm language must be expressive and enable users to write efficient source code that is easily understandable with minimal syntax. It must also produce readable source code within the target simulation platforms. The translated program must be a standalone program that is independent of the transformation system.

**(*v*) *Validate correctness using unit tests.*** Given that CyML is built to serve as an intermediate representation of a set of languages, its validity is practically proved if all unit tests written in CyML succeed in all languages after transformation. This involves testing the generated code either in a multilanguage runtime environment or in the runtime environment of each language to ensure that the language features are well defined and that their emulation in other languages is correct.

To satisfy the above requirements, we identify common patterns often used in crop modeling simulation platforms to implement model components. They result from the intersection of a set of minimal features of different languages used by the platforms (Figure 1, left part). We used these features to propose a shared modelling language. An additional design choice is to use a subset of an existing language that can satisfy our requirements and provide the common selected features. Python was a good candidate language to fit our design considerations. It is an expressive and high-level programming language that allows writing short source code and has a gentler learning curve than C, C#, Java, or C++ (Linge and Langtangen 2016). However, its dynamic typing can make transformation into programming languages with static typing ambiguous. Therefore, we proposed to add an explicit type declaration to the Python language, which led us to choose Cython (Behnel et *al.* 2000). Cython is a high-level programming language that combines the power of Python and **C function** calling and **types** on variables and class attributes. It is compiled directly in efficient C code that improves runtime speed and allows it to interact with C, C++ and Fortran source code. However, not all Cython syntax can be directly translated into all target languages. For instance, the yield statement and anonymous functions are not supported by Fortran. Therefore, we defined CyML as a sub-set of Cython to address the implementation of the model algorithm (Figure 1, right part). CyML does not cover some features such as class definition, nested functions, exceptions handling, anonymous function, reading and writing files. These features are handled by the platforms in their programing language.

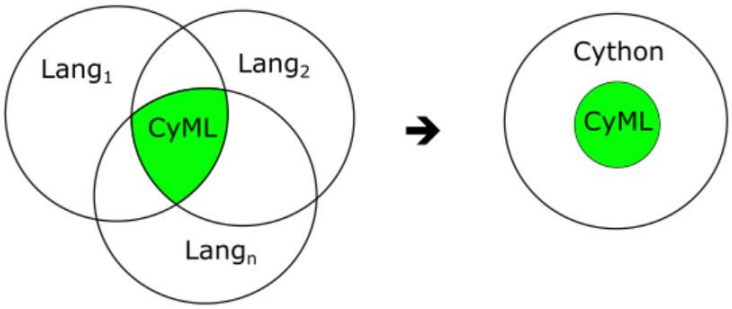



**Figure 1.** From the intersection of a set of languages features to a definition of an abstract language CyML, defined as a subset of Cython. $Lang_i$ corresponds to a minimal language supported by a crop simulation platform "i". The number of circles (n) in the left corresponds to the number of platforms.

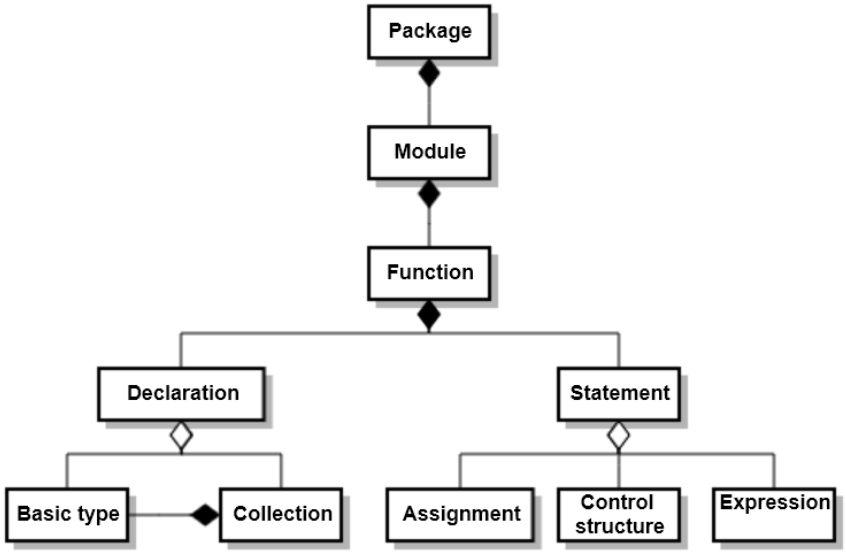



**Figure 2.** Main concepts supported by the CyML language. Black diamonds indicate composition ("contains") relationships and white diamonds indicate a specialization ("is-a").

## 2.3 CyML language

CyML is designed as a subset of the Cython language based on a language specialization approach. This involves removing undesirable syntactic or/and semantic features of Cython that may not be easily transformed into many different languages or are not required to implement PBM algorithms. The conformance to the subset of Cython features is guaranteed through a semantic analysis. The main concepts supported by CyML are represented in Figure 2.

***Declaration: Basic types and collection.*** Unlike CyML, Cython does not require explicit type declarations. This means that in CyML, all variables have to be declared before they are used and the declared type is immutable. A variable can be initialized during or after its declaration. In the case of model algorithm implementation, a variable can be either a model input, output or a local variable required for the implementation. Explicit static typing is enforced by the semantic analysis step illustrated in Figure 2. CyML supports basic types (e.g. integer, real, logical and string) and two sequence types (list and array) with dynamic or fixed length. Each element of a sequence must have the same type. Moreover, since time is an important variable in the defintion of discrete-time process, CyML provides datetime types in terms of year, month, day, hour, minute and second. CyML suppports commonly used binary (numerical and boolean), unary and comparison operators, as well as casting operators for basic types and sequence operators such as length or sum.

***Statements***. Statements can be either an assignment, an expression or a control structure. An *assignment* assigns a variable to a mathematical expression, another variable or a value using an assignment operator (e.g. "="). An assignment statement can, therefore, express the relationships between model inputs-outputs when those are described only by simple equations. An *expression* is commonly defined as a construct made up variable, operator, or function call that can be evaluated to a value. In CyML, expression is distinguished from assignment by the fact that, in the case of assignment construct, the evaluation result of an expression is assigned to a variable. An *expression* can contain standard mathematical functions such as exponential, maximum, minimum, and power functions. Unlike assignment, expressions have no assignment operator. They are built-in functions called to perform an operation (e.g. collection operations such as adding or removing an element in a sequence). CyML supports structured control flow statements that can be nested. Control flow statements include conditional branching (if, elseif, and else) and loops (for-in-range, for-each, iterating over several collections, and while) statement.

***Function.*** CyML uses the definition of a Python function to code the model algorithm and to represent external functions with arguments with explicit data types. A function is composed of a set of statements in its body grouped under a *def* statement with a signature consisting of the name of the function, their inputs arguments and return values. A function may call other functions that can be provided by an import mechanism to ensure modularity. CyML also supports recursion which means that a function can call itself in its definition.

***Module and package*.** A module is a file containing a set of functions that can be reused in models and functions. A package contains a set of modules and models in a set of files. These concepts allow external dependencies to be managed.

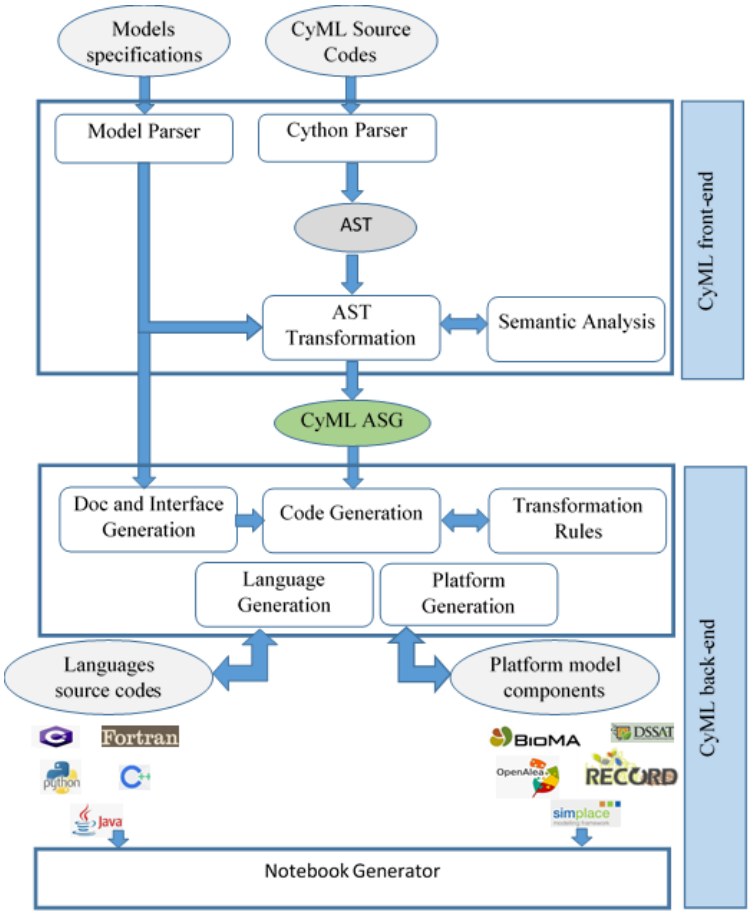



**Figure 3.** Design architecture of the one-to-many CyML transformer (CyMLT). It takes as input a model unit algorithm implemented in CyML with associated model specifications and applies a transformation workflow to produce crop model components or source code in different languages for different platforms.

## 2.4 CyMLT design

The CyMLT architecture is composed of two main parts: the *front-end* and the *back-end* (Figure 3).

The *front-end* consists of a *Model Parser*, a *Cython Parser*, and a *Semantic Analysis* component.

The *Model Parser* checks the model specification based on the Crop2ML grammar and generates a logical object allowing access and manipulation of the model.

The *Cython parser* provides a lexical and syntactic analysis of the source code. It detects syntactic errors and generates an *Abstract Syntax Tree* (AST). The AST is a data structure representing the syntactic structure of the source code as a tree where the nodes represent the syntactic components (e.g. FunctionDefinition, Assignment, If-Block…) of the grammar. Figure 4 shows an example of AST generated from a square function. The design choice of CyML relies on the legacy Cython parser. This parser uses all the syntactic components of Cython instead of a restricted grammar. To restrict Cython grammar, the generated Cython AST is processed to ensure that it incorporates only syntactic components defined in CyML.

The *AST Transformation* transforms the generated AST to a self-contained representation of the source code called *Abstract Semantic Graph* (ASG), which is independent of the source language.

The *Semantic Analysis* operates during the AST transformation to perform semantic checks from the AST. It consists of various checks such as type consistency, declaration of variables before their use, or consistency of elements in a list. This analysis checks that the input and output datatypes in model specifications are well defined in relation to the model algorithm. The semantic analysis generates error messages if the verification fails. Note that, unlike the AST, each node of the ASG is labeled with at least its type and its pseudo-type (Figure 4c). The pseudo-type is the expected type of a node and strengthens code generation reducing the number of ASG traversals. For example, in Figure 4c a node of type “Function” follows “Module node” and has a pseudo-type [“Function”, “int”, “int”]. This pseudo-type corresponds to the function signature, meaning that this function takes as input one argument of type “int” and returns one value of type “int”. Note also that, unlike the AST, the type of internal nodes of the ASG may be different from non-terminal symbols of the grammar. Another type of node is built that preserves the intention in the source code instead of the code structure. For example, in Figure 4b the binary operator node “PowNode” is transformed in Figure 4c by a “standard call” node, which takes as arguments the operands of the binary operation.

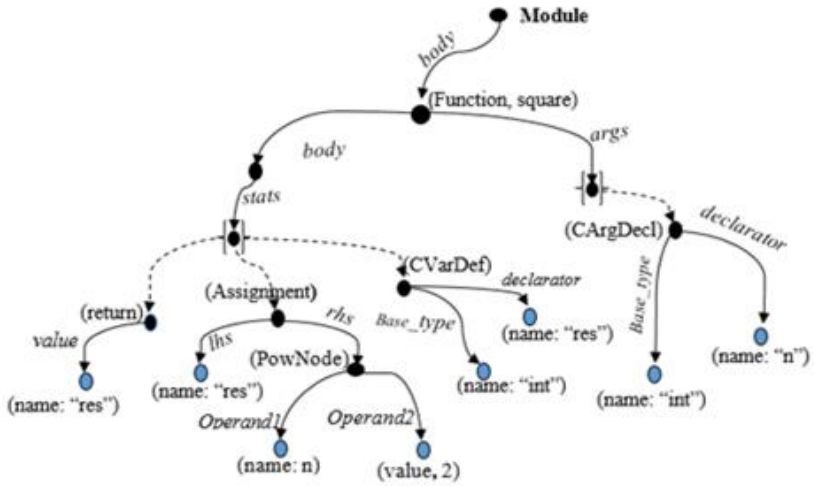
The back-end of CyMLT is responsible for *Code Generation (*Figure 3*)*. It is independent of the front-end. It takes as input the ASG generated by the front-end and works in relation with the *Doc and Interface Generation* and *Transformation Rules* components.

The *Code Generation* component transforms the annotated ASG into different readable source code or platform components. It consists of two integrated sub-components: a *Language Generation* and a *Platform Generation*. A L*anguage Generation* emits the source code in a specific language with a specific programming paradigm. This source code does not contain any simulation platform features. A *Platform Generation* emits a model component based on the requirements of a platform such as its implementation language, software design and code conventions.

(a)

```
1 def square(int n):
2     cdef int res
3     res = n**2
4     return res
```

(b)




(c)

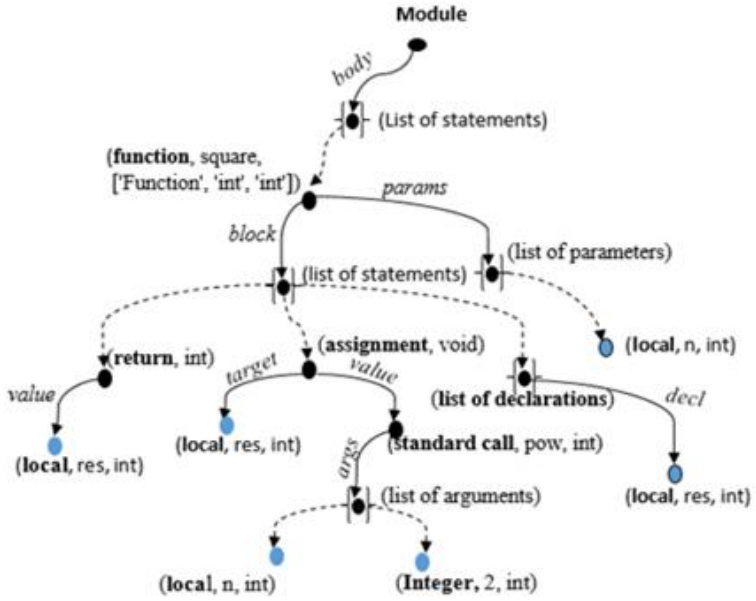



**Figure 4.** Example of abstract syntax tree (AST) and abstract semantic graph (ASG). (a) definition of function "square" in CyML. (b) simplified view of AST of function “square” where the internal nodes in black represent Cython constructs and the final node in blue a variable or constant. (c) Simplified view of ASG with of function “square” with the new annotated nodes. The leaf nodes in black are non-terminal symbols of the Cython grammar whereas the end blue nodes are terminal symbols, essentially variables and constants. A child node (c) can be accessed from its parent node (p) through an attribute ($p\ c$).

A *Transformation Rule* is a function that takes as input a node of the ASG and generates a new node based on a specific structure of the target language. *Transformation Rules* are applied on the ASG for *Code Generation.* The code generation is generally described by straightforward transformations of the ASG. However, some nodes of the ASG require non-trivial transformations to produce new nodes. For example, the transformation of the declaration node in Figure 4c consists of replacing the basic type `int` by the Java basic type `integer` without the `cdef` statement to reproduce Java integer variable declaration, whereas the generation of the power call function requires applying a casting function (`int`) to preserve type compatibility.

The *Doc and Interface Generation* component generates documentation in the target language from the model specification. It embeds all the semantics of model inputs and outputs, and then integrates the model knowledge in the code generated.

Finally, the *Notebook Generator* transforms generated source code or model components into Jupyter notebook (Kluyver *et al.* 2016) to interactively test and validate the transformation.

## 2.5 CyMLT implementation

CyMLT proposes a unique approach to transform an ASG into many programming languages. It is implemented around the main classes shown in Figure 5. A set of classes (suffixed by *Generator*) generates the code for each language and platform. It means that a sub-class of *PlatformGenerator* and of *LanguageGenerator* class have been implemented for each supported platform and language. A *PlatformGenerator* class inherits attributes and properties of the *LanguageGenerator* class related to the language used by the platform. For example, as BioMA uses the C# language, the *BioMAGenerator* class (i.e. the class that generates BioMA components) inherits the *CsharpGenerator* class that generates the source code in *C#*. Each class contains a visitor method for each ASG node type. Each visitor method name is composed of "visit_" followed by "the type of the node". A visitor method emits code fragments. Each *LanguageGenerator* sub-classes provide the same visitor method names given that the same ASG is used. A *LanguageGenerator* class also inherits two classes: *CodeGenerator* and *LanguageRule*. The *CodeGenerator* class contains the factorized methods shared by all *LanguageGenerator* classes including the method used for code emitting and code formatting. This class inherits the super class of the transformation process called *NodeVisitor*. CyMLT implements the Visitor design pattern (Gamma *et al.* 1995) to avoid a procedural implementation approach. *NodeVisitor* contains a dispatch method that enables recursive traversal through the nodes. During traversal, the appropriate visitor method corresponding to the type of the current node is called in *LanguageGenerator* or *PlatformGenerator* and the associated code fragment is emitted. Before emitting the code fragment, some nodes undergo a transformation from the

*LanguageRule* class. This class is implemented for each language as a mapping where keys corresponds to the different methods, datatypes, and operators of CyML, and values are their emulation in target languages provided from their standard libraries (Supporting Information Table S1 to S5). Given that the CyML language is similar to Python, it is straightforward to yield Python code through one ASG traversal. This is not the case for all target languages, which require more traversals to support specific features provided from the analysis of the ASG. For example, a first traversal could detect that it is necessary to declare other variables in the generated code. These additional operations have been implemented in the *Adapter c*lass containing some methods to traverse the ASG and, where the conditions have been defined, to retrieve the new features required in *LanguageGenerator*. Likewise, the *Model* object generated by the model parser is used in *LanguageGenerator* to generate the model interface with accessor and mutator methods for object-oriented languages, or to add additional semantics to variables based on platform conventions. This separation of model specification from model algorithm enhances CyMLT to transform a model algorithm from a procedural approach to an object-oriented approach with different software designs. Finally, *LanguageGenerator* and *PlatformGenerator* use *DocGenerator* to integrate model documentation into generated model components. *DocGenerator* extracts all information based on model specification and presents it in different format according to the language and the platform.

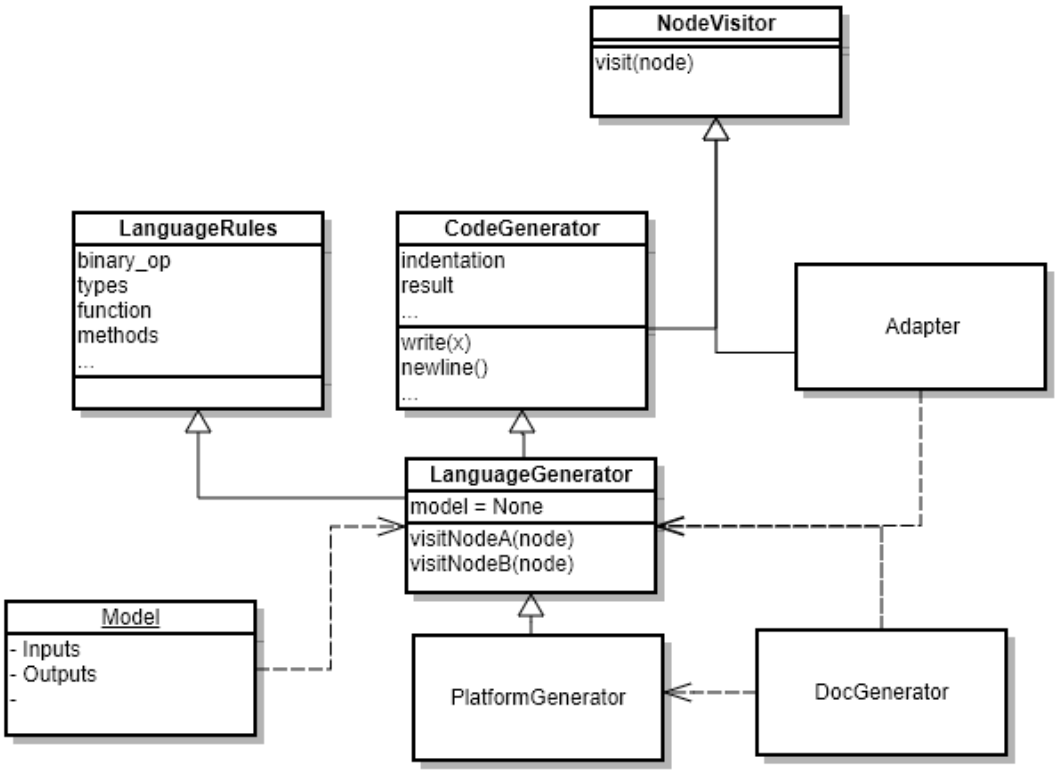



**Figure 5.** Class diagram illustrating the implementation of the one-to-many CyML transformer (CyMLT).

## 2.6 Case study

Phenology, the timing of crop development and the simulation of phase durations and crop stages, is sometimes thought of as the core for most crop growth PBMs and an essential component of most crop modeling platforms. In order to illustrate how a model is written in CyML and the functionalities of the language, we transformed the BioMA phenology component (Manceau and Martre 2018) of the wheat PBM *SiriusQuality* (He *et al.* 2012) into a Crop2ML composite model and wrote the algorithms of the model in CyML. The *shootnumber*, a model unit of this component, is presented in Supporting Information Listing S1.

# 3. Results

## 3.1 Model algorithm implemented in CyML

The *shootnumber* model is implemented in CyML as a function that includes all the meta information provided by the model specifications (Supporting Information Listing S2). The model documentation is generated from the model specification and is shown in red. It contains the name of the model, its version, its time step (in days) and other descriptions such as the authors' names and the reference for the model.

The algorithm *shootnumber* unit model requires an external function, Fibonacci, which is implemented outside of the model algorithm (Supporting Information Listing S2, Line 35) to make the code readable and shorter. This mathematical function allows to compute the shoot production from the number of emerged leaves on shoots (Supporting Information Listing S2, Line 22). We implement the code using conditional (if, line 26) and loop (for, line 29) control structures. Table 1 gives the meaning of CyML language built-in functions that are used to implement the *shoot number* model.

**Table 1.** Example of built-in functions within CyML language and their meaning.

| Function | Description |
|---|---|
| max | Largest item in a sequence |
| min | Smallest item in a sequence |
| ceil | Smallest integer greater than or equal to the parameter |
| append | Add an element at the end of a dynamic array (list) |
| len | Number of elements in a sequence (array or list) |
| range | Generate a list of integers from a start value to a stop value with a step |
| integer | Update the actual state variable from its previous value and the rate |

## 3.2 Transformation of CyML source code to different languages and platforms

Currently, CyMLT supports Python, Java, C#, C++ and Fortran languages. It also has the capability of generating a model algorithm in conformance with crop simulation platforms requirements. Therefore, it handles different programming paradigms such as procedural, functional, and object-oriented programming by associating model specifications to the transformation workflow.

**Structure of generated source code.** Although CyML provides a procedural mechanism to implement model algorithm, the programming languages supported by CyMLT can be classified in procedural and object-oriented programming paradigms. Some languages are designed to support only the object-oriented paradigm (C# and Java). Fortran and C are procedural languages even though they can "mimic" some object-oriented features to support object-oriented programming style (Cary *et al.* 1997). Python and C++ support both object-oriented and procedural paradigms. CyMLT uses procedural paradigm for Python and object-oriented for C++, as these are the most often used approaches in these languages. However, CyMLT can also be extended to generate models in Python with an object-oriented approach and in C++ with a procedural approach.

For the C++, C# and Java languages, a model algorithm implemented in CyML is transformed into a class (Listing 1) that encapsulates both the algorithm and the scientific knowledge related to the model through the integrated documentation. A class, in software engineering terms, is a data structure defining a set of common properties and methods of an object. The generated source code contains methods to access and mutate model inputs and outputs, a constructor method to create and initialize an instance of the model (object) and a calculation method encapsulating the procedural logic of the model algorithm. First, variables are used to access model input (Listing 2) values before transforming the set of instructions of the model algorithm into the new language. Then, mutator methods are applied to update the model outputs (Listing 3). Model inputs and outputs are used to

build a class of objects passed in argument of the calculation method. External functions are transformed into static methods of the model class (Listing 1**Error! Reference source not found.**).

Java

```java
public class Shootnumber
{
    private double sowingDensity;
    public double getsowingDensity()
    { return sowingDensity; }
    public void setsowingDensity(double _sowingDensity)
    { this.sowingDensity = _sowingDensity; }
    ...

    public Shootnumber() {...}
    public void  Calculate_shootnumber(...)
    {
        ...
    }
    public static int fibonacci(int n)
    {
        ...
    }
}
```

C#

```csharp
public class Shootnumber
{
    private double _sowingDensity;
    public double sowingDensity
    {
        get { return this._sowingDensity; }
        set { this._sowingDensity = value; }
    }
    ...
    public Shootnumber() {...}
    public void  Calculate_shootnumber(...)
    {
        ...
    }
    public static int fibonacci(int n)
    {
        ...
    }
}
```

Fortran

```fortran
    SUBROUTINE model_shootnumber(
        sowingDensity, &
        ...)
        IMPLICIT NONE
        REAL, INTENT(IN) :: sowingDensity
        ...
    END SUBROUTINE model_shootnumber

    RECURSIVE FUNCTION fibonacci(n) RESULT(res_cyml)
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: n
        ...
    END FUNCTION fibonacci
```

C++

```cpp
class Shootnumber
{
    private:
        float sowingDensity;
        ...
    public:
        Shootnumber();
        void  Calculate_Model(...);
        int fibonacci(int n);
        float getsowingDensity();
        void setsowingDensity(float _sowingDensity);
        ...
};
```

**Listing 1.** Structure of generated source code in Java, C#, Fortran, and C++.

The current version of CyMLT supports Fortran 90. This Fortran version presents low-level features (pointers, allocation), which makes some transformations difficult but ensures a higher portability. In Fortran, model algorithm corresponds to a subroutine, whereas external functions are subroutines, functions or recursive functions. CyMLT automatically operates this choice. In our case study, the Fibonacci function is transformed in a recursive function, which keeps the structure of the original code. In Python, the generated source code has the same structure as the CyML function. However, CyMLT can also generate Python code with an object-oriented approach.

```
126         double canopyShootNumber_t1 = s1.getcanopyShootNumber();
127         double leafNumber = s.getleafNumber();
128         List<Double> tilleringProfile_t1 = s1.gettilleringProfile();
129         List<Integer> leafTillerNumberArray_t1 = s1.getleafTillerNumberArray();
130         int numberTillerCohort_t1 = s1.getnumberTillerCohort();
```

**Listing 2.** Access input variables (in Java), s and s1 correspond to two instances of the class of state variables to manage previous and current state. CyMLT generates variables to access the fields of these instances and uses them in the procedural logic.

```
156         s.setaverageShootNumberPerPlant(averageShootNumberPerPlant);
157         s.setcanopyShootNumber(canopyShootNumber);
158         s.setleafTillerNumberArray(leafTillerNumberArray);
159         s.settilleringProfile(tilleringProfile);
160         s.setnumberTillerCohort(numberTillerCohort);
```

**Listing 3.** Update output variables in Java. s corresponds to an instance of current state variable.

***Data type and variable declaration.*** In addition to the programming paradigms, languages supported by CyMLT can be classified by their type system, in particular their type expression (explicit or implicit). This can affect the quality of the generated code. Although some languages (e.g. C# and C++) allow both implicit and explicit type expression, we chose to provide explicit typing. Basic types (integer, logical, character, and real) are built-in data types in all languages. However, other more complex types like `datetime` or `sequence` are supported but require external or standard libraries. Moreover, various libraries exist to handle the same data structure. CyMLT's datatypes map appropriately to target languages by using their standard library (Supporting Information Table S1).

Some compromises have been made for the transformation of complex types. CyML arrays are modeled on a standard Python list. However, the size of list datatype variables is not fixed. We propose to use the Numpy array in the next version of CyMLT. In Fortran, CyMLT generates allocable arrays to map to CyML list data types and provides some functions to handle it. These functions are extracted from CyMLT library and integrated into the generated code to make it independent of the library of transformation. In C++, `datetime` type handling is not easy. It is converted into a string, which could be split for processing. CyML arrays without a specified size in the function parameter are mapped to C++ arrays using templates (Listing 6, line 1). In Java, there are many standard Time APIs. (e.g., `Date`, `LocalDateTime`) depending on the version of Java. We have chosen to use the `Date` Library in Java and the `DateTime` Library in C#.

**Type and intent preservation.** Most of the target languages provide built-in methods matching with CyML built-in functions. However, there may be some differences between their name or return types. This is considered in the generated source code. As an example, consider the statement at

**Error! Reference source not found.** on Line 29, where the purpose is to find the smaller integer value that is larger than or equal to the leaf number. The method *ceil* in the C++ Math library corresponds to the CyML *ceil* function but returns a floating-point value. In this case, CyMLT preserves the original type (integer) by applying an explicit type conversion (Listing 4, Line 1).

```
1    for (i=leafTillerNumberArray_t1.size() ; i<(int) ceil(leafNumber) ; i+=1)
2    {
3        lNumberArray_rate.push_back(numberTillerCohort);
4    }
```

**Listing 4.** Type preservation in CyML transformation to C++, int casting is applied to the result returned by *ceil* function.

The generated code preserves the intent of the original code provided by the information on the ASG. Listing 5 illustrates this intent preservation in the transformation of CyML `For-loop` construct (Listing 4, Line 1) where the consecutive iteration is expressed into an efficient way of representation in Fortran with the *DO sequence* (Listing 5, Line 1). However, the sequence indexing is different between CyML and Fortran. The last parameter of the CyML range function is not contained in the CyML sequence unlike the Fortran *DO* sequence. This is managed by subtracting this parameter by 1 in the generated code, thereby providing a same length of sequence. Likewise, arrays in Fortran are indexed from 1 by default and this is considered during the transformation of all array operations.

```
1        DO i = SIZE(leafTillerNumberArray_t1) , CEILING(leafNumber)-1, 1
2            call Add(lNumberArray_rate, numberTillerCohort)
3        END DO
```

**Listing 5.** From CyML for-loop to Fortran do-loop. The subroutine Add is generated to expand leaf tiller number array.

**Preservation of the scope of variables.** CyMLT considers the scope of the variables in the different target languages. The scope of a variable refers to a region of the code where the variable is visible. Some languages like Java, C++ and C# manage variable scope differently and this variability is handled by CyML.

Consider the transformation of a simple CyML function that calculates the sum of elements of an array x with undefined size (Listing 6). The generated code in Fortran requires the declaration of new variable *`i_cyml`* to map the For-loop construct. However, the generation of a new variable in Java, C++ and C# preserves the scope of the variable `i`. The scope of the iteration index on an array variable in a `For-loop` construct is limited to the loop scope, whereas it is extended to all the functions in CyML and Python. Assuming that in the original code this iteration index is reused after

the loop, it will generate a compilation error in the target languages if the transformation did not handle this scoping issue by declaring another variable.

CyML

```
def sum_(int x[]):
    cdef int i, y=0
    for i in x:
        y = y + i
    return y
```

Python

```
def sum_(x):
    y = 0
    for i in x:
        y = y + i
    return y
```

C++

```
template<size_t SIZE_0>
int sum_(const array<int, SIZE_0>& x)
{
    int i;
    int y = 0;
    for(const auto& i_cyml : x)
    {
        i = i_cyml;
        y = y + i;
    }
    return y;
}
```

C#

```
public class Test
{
    public static int sum_(int[] x)
    {
        int i;
        int y = 0;
        foreach(int i_cyml in x)
        {
            i = i_cyml;
            y = y + i;
        }
        return y;
    }
}
```

Java

```
public class Test
{
    public static int sum_(Integer [] x)
    {
        int i;
        int y = 0;
        for(Integer i_cyml : x)
        {
            i = i_cyml;
            y = y + i;
        }
        return y;
    }
}
```

Fortran

```
FUNCTION sum_(x) RESULT(y)
    IMPLICIT NONE
    INTEGER , DIMENSION(: ), INTENT(IN) :: x
    INTEGER:: y
    INTEGER:: i
    INTEGER:: i_cyml0
    y = 0
    DO i_cyml0 = 1, SIZE(x)
        i = x(i_cyml0)
        y = y + i
    END DO
END FUNCTION sum_
```

**Listing 6.** CyML code of a function that computes the sum of the elements of a list transformed using CyMLT in Python, C++, C#, Java, and Fortran.

## Transformation to simulation platforms

The transformation of a CyML code to target languages can generate a model component in different ways. These transformations have been designed to be close to the philosophy of each target language. However, from the perspective of crop model component development, high-level

programming languages are the lowest level of abstraction with respect to simulation platforms and frameworks. Additional constraints in crop modeling platforms include a specific programming paradigm, software design and code conventions. These different features give them capabilities to provide code introspection and reflection support, which allows them to dynamically extract and change information or knowledge about the code at run time. Thus, the code generation should extend language code generation by considering platform coding constraints, which are often implicit. The design of programming languages is formalized using grammars and is unambiguous. Platforms use design and architectural patterns without the use of an explicit formalism. This implies adapting the transformation to each platform taking into account their specificities. The current version of CyMLT generates model components compatible with BioMA, DSSAT, Record, OpenAlea and Simplace platforms, which support C#, Fortran, C++, Python and Java, respectively.

***Generation of object-oriented components.*** An object-oriented platform provides features such as inheritance, polymorphism and software design used to implement models. Polymorphism allows a model programmer to provide a generic interface to a number of related functions, and, thus, to propose different strategies to implement a model with different assumptions. For instance, this provides the possibility to include new physiological processes that are shared among different crop types. For this, object-oriented platforms define an abstract class that specifies the interface of all model components, which implements all the abstract methods of the abstract class. Two different approaches are used for model components to inherit an abstract class. Some platforms offer an abstract class and all model components implement and extend this class. This is the case for Simplace and Record, which provide the FWSimComponent (Listing 7: Structure of ShootNumber component in Simplace. A model unit in Simplace implements and extends an abstract class called FWSimComponent. Then, a model component overrides its abstract methods including init (model initialization), clone (deep clone of the model) and process (model algorithm). The structure of the abstract class is used to define a model skeleton in CyMLT to generate a model conforms to platform requirement.) and *DiscreteTimeDyn* interface, respectively. Another approach followed by platforms is component-based programming. A model developer creates a component that inherits of an interface provided by the platform. Thus, model components inherit this component interface. For example, BioMA provides the IStrategy interface. The current version of CyMLT generates a component interface in addition to the generation of model components. The abstract methods depend on the platform and include a method that encapsulate the algorithm of the model.

```java
public class Shootnumber extends FWSimComponent
{

    // Constructor
    public Shootnumber(){
        super();
    }

    @Override
    protected void process()
    {
        // model algorithm
    }

    @Override
    protected void init()
    {
        // Component initialization
    }

    @Override
    protected FWSimComponent clone(FWSimVarMap aVarMap)
    {
        // creates a clone from this Component for use in other threads
    }

}
```

**Listing 7**: Structure of ShootNumber component in Simplace. A model unit in Simplace implements and extends an abstract class called FWSimComponent. Then, a model component overrides its abstract methods including *init* (model initialization), clone (deep clone of the model) and process (model algorithm). The structure of the abstract class is used to define a model skeleton in CyMLT to generate a model conforms to platform requirement.

***Generation of stateless and stateful unit models.*** A model algorithm is implemented in CyML as a function. However, the CyMLT generates both a stateless and a stateful component. A stateless component is an immutable object whose values of fields do not change if methods are invoked. CyMLT allows searching and extracting state variables from a model specification to perform code generation according to each platform.

In DSSAT and OpenAlea, a model algorithm is implemented as a stateless functional component (declarative paradigm). The Fortran code generated by CyMLT is compatible with DSSAT. In this platform, the calculation of rates of change and the integration of state processes are sometimes separated with the use of a control variable. In CyML, we introduce two variables that define the previous and current value of a state variable that avoids a misuse of the state variable. Although OpenAlea offers capabilities to benefit of oriented-object features of Python, OpenAlea components can be defined as pure Python functions, already generated by CyMLT. However, model specifications need to be transformed into an OpenAlea component specification for unit and composite node (Pradal *et al.* 2008).

BioMA uses the strategy design pattern to create a library of simple strategies (equivalent to Crop2ML unit models) and composite strategies for model composition. The simple strategy leads to the implementation of a model unit as a stateless component. Thus, an instance of model unit class is a stateless object since it contains only model parameters (if any) as attributes which do not change during the simulation. The method of computation is comparable to a function that takes an object as an argument (i.e. higher-order function). Concretely, these objects are instances of domain classes. Domain class contains the values and the attributes for all variables defined in model specifications. To handle the change of state variables, the method of computation of each class takes as arguments two instances of state variables domain class reproduced by CyMLT (Listing 8), one for the current value and the other one for the previous one. This is made possible by the fact that the previous state is emulated in the CyML function with variable suffixed with "_t1".

Finally, in Record and Simplace, unlike BioMA, a model unit class contains all state variables. In Simplace, there is no convention to distinguish previous and current state variables. Thus, CyMLT considers them as distinct fields in the generated Simplace component. The Record platform handles variable history (time series) by suffixing state variable with an operator `()` in the code. Thus, in this case, CyMLT generates current state variables with the suffix `()` and previous state variables with `(-1)`.

```
20      public void  Calculate_shootnumber(State s, State s1, Rate r, Auxiliary a)
21      {
22          ...
23          double canopyShootNumber_t1 = s1.canopyShootNumber;
24          double leafNumber = s.leafNumber;
25          List<double> tilleringProfile_t1 = s1.tilleringProfile;
26          List<int> leafTillerNumberArray_t1 = s1.leafTillerNumberArray;
27          int numberTillerCohort_t1 = s1.numberTillerCohort;
28
29          ...
30
31      }
```

**Listing 8.** Fragments of code in C# with BioMA guidelines generated with CyMLT. S1 is an instance of state domain class used for previous time, s is an instance of state domain class used for current time. This shows that leaf number has been calculated by another model at the current time step, whereas the other variables are those calculated at the previous time step.

**Generation of platform specific types and data-structures.** Some platforms define their own types by providing a generic class to handle model variables and parameters. A generic class is either a class or an interface that can be parameterized over the language data types. It contains a specific

number of methods including methods to access or update variables. In this case, CyML data types map the framework generic types.

Unlike BioMA, where inputs and outputs are C# data types extended with the generation of accessors and mutators, Simplace and Record provide their own class or interface to declare model inputs and outputs. To generate a Simplace component, the process of transformation consists of declaring model variables with the specialized class *FWSimVariable*. Then, CyMLT generates other variables declared with Java data types, which are used to access values of the *FWSimVariable* instances (Listing 9). This allows expressing the model algorithm with a pure Java but requires the use of a mutator method of the generic class to update output (Listing 10). Likewise, the generated Record component implements the *DiscreteTimeDyn* class provided by the vle package of Record to encode discrete-time models algorithms.

```
42          double tcanopyShootNumber_t1 = canopyShootNumber_t1.getValue();
43          double tleafNumber = leafNumber.getValue();
44          double tsowingDensity = sowingDensity.getValue();
45          double ttargetFertileShoot = targetFertileShoot.getValue();
46          List<Double> ttilleringProfile_t1 = Arrays.asList(tilleringProfile_t1.getValue());
```

**Listing 9.** Generation of other variables to access Simplace component variables. These variables are prefixed by t.

```
74          averageShootNumberPerPlant.setValue(taverageShootNumberPerPlant, this);
75          canopyShootNumber.setValue(tcanopyShootNumber, this);
76          leafTillerNumberArray.setValue(tleafTillerNumberArray.toArray(new Integer[0]), this);
77          tilleringProfile.setValue(ttilleringProfile.toArray(new Double[0]), this);
78          numberTillerCohort.setValue(tnumberTillerCohort, this);
```

**Listing 10.** Update of the variables of the shootnumber unit model generated by CyMLT following Simplace specifications.

## 3.3 Extensibility

The number of languages and platforms that CyMLT supports can be extended due to its modular structure. The explicit separation between the production of the annotated ASG and its transformation into a readable source code of the target languages and platforms provides a great flexibility to add new target languages. The addition of a new language requires only a mapping of this intermediate representation into a set of compatible instructions based on the standard library of the language. The generated code must be independent of the transformer, clear, and easy to read while preserving the knowledge expressed in the original code. We present the steps for the extension of CYMLT with R language (R Core Team 2017) and the Plant Modeling Framework (PMF).

**Supporting a new language: R.** R is a popular language used for statistical analyses and data visualization. Many modelers use R to start the development of their model (Zhao *et al.* 2019). Thus, with this extension, modelers can in the same environment conduct the first steps for model development and the implementation in a simulation platform, and analyze model outputs. The extension of CyMLT for R relies on the implementation of *RGenerator* and *RRules* classes that emit fragments of code in R and define transformation rules between CyML and the desired R constructs, respectively.

**Implementation of transformation rules for R.** Transformation rules define the mapping of CyML operators, built-in functions and methods to their equivalent in R. R is a dynamic typed language and, as with Python, the type of variables is ignored.

*Operators mapping*. Listing 11 declares the mapping between CyML and R operators. Only the difference operators are shown between CyML and R. During the ASG traversal, the `visit` method considers these mappings to emit code fragments.

```
binary_op = {"and": "&&",
             "or": "||",
             "not": "!",
             "%":"%%",
             ...
             }
```

**Listing 11.** Operators mapping.

*Adapting Standard Functions.* CyML defines three standard libraries (i.e. math, system, and io) to provide mathematical, system, and file management functions in the different languages. A mapping is needed to link these functions to native R ones for each library. Some functions are identical between CyML and R, like `min` or `max`. Others require a transformation to another type of node. It is useful for model developers to observe the generated ASG of each CyML construct in order to define the equivalent of the construct. For example, the construct of a `modulo` binary operation in CyML is a *standard_call* node in the ASG whose namespace is `system`, the function is `modulo` and the arguments are the two operands. This node is transformed into a binary_op node (binary operation) with the function “translateModulo” (Listing 12). The new node is visited to produce R fragment code.

```
functions = {
    'math': {
        'ln':           'log',
        'log':          'log',
        'tan':          'tan',
        ...
    },
   'io': {
        'print':    translatePrint,
         ...
    },
    'system': {
        'min': 'min',
        'modulo': translateModulo,
         ...
    }
}

```

**Listing 12.** Standard functions mapping.

*Standard methods mapping.* Standard methods are functions applied to a particular data type of CyML language (Listing 13). Thus, a set of methods is provided for each CyML datatype. Their equivalents in R language are defined using the same mapping mechanism used for standard functions. In Listing 13 at Line 9 the `append` method applied to a list is transformed to an assignment node whose value is a function `c` that takes as arguments the name of the variable of type list (receiver) and the argument of the `append` method (args). The definition of these rules limits the use of conditional statements in the implementation of the visit methods and facilitates the extension of CyMLT.

```
methods = {
    'int': {
        'float': 'as.double',
        ...
    },
    ...
    'list': {
        'len': 'length',
        'append': lambda node: Node(type="assignment", target=node.receiver,
                                    value=Node(type="call", function="c",
                                               args=[node.receiver,node.args])),
        ...
    }
}
```

**Listing 13.** Standard methods mapping.

**Implementation of a R code generator**. The *RGenerator* class inherits the *RRules* class. It implements a family of `visit` methods like `visit_assignment`, `visit_bool` related to all types of nodes provided by the ASG. These methods emit fragments of code, which will be joined to produce a formatted source code in R. The properties that enable write and format functions for these

fragments are implemented in a class named *CodeGenerator* inherited by *RGenerator*. Additionally, *CodeGenerator* abstracts the common behavior of these languages by providing other properties and `visit` methods common to all the target languages. Some methods are redefined in the language generator when it has particular features. The developer of the R code generator implemented the different visit methods without bothering with the dispatching mechanism provided by the NodeVisitor class. A `visit()` method is called for all composite child nodes while a *write()* method is invoked for the terminal or single node to emit the code fragment. For example, a boolean value is a terminal node. Thus, the `visit_bool` method allowing generation of the corresponding boolean value in R will only consist in uppercase CyML logical value (Listing 14).

```
1       def visit_bool(self, node):
2           self.write(node.value.upper())
```

**Listing 14.** Implementation of logical value transformation.

The assignment node is a composite node that contains a *target* node and a *value* node. These two nodes could be a composite node. So, they will all be visited by the `visit_assignment()` method (Listing 15).

```
1       def visit_assignment(self, node):
2           self.newline(node)
3           self.visit(node.target)
4           self.write(' <- ')
5           self.visit(node.value)
```

**Listing 15.** Implementation of assignment transformation.

All target language generators share the principle of implementing a visitor method for standard functions or standard methods call nodes, and, it is, therefore, implemented in the *CodeGenerator* class. The properties of the node are used to access to the function equivalent in the dictionary of functions in the transformation rules class.

Listing ***16*** shows the implementation of the standard function call node where its properties such as namespace and function are used to access the equivalent function.

```
    def visit_standard_call(self, node):
        node.function = self.functions[node.namespace][node.function]
        self.visit_call(node)
```

**Listing 16.** Implementation of standard function call.

This implementation approach is followed for all types of nodes and could be gradually done according to the expected R constructs. Given that it has several possibilities to implement an algorithm, it is the responsibility of the extension developer to provide the corresponding semantic for each particular node of the ASG and to validate the transformation with unit tests.

**Supporting a new simulation platform: APSIM-PMF.** APSIM (Holzworth *et al.* 2014) is one of the most widely used PBM platforms for simulating the performance of a wide range of cropping systems. It has undergone a major evolution by providing the Plant Modelling Framework (PMF; Brown et *al.* 2014). PMF is used to build models that represent plant components of a crop composed by identical plants. It is based on the structure of a generic plant and a wide range of processes involved in plant growth and development. However, the composition and parametrization to build a particular crop model is not specified and is left to model developers. PMF, therefore, allows great flexibility in its approach for implementing biophysical processes by separating model set up and assembly. The PMF concepts and processes are implemented as generic classes at different organizational levels (Brown et *al.* 2014).

The extension of CyMLT to PMF consists in adding the capacity to generate a model component in C# that fulfills PMF requirements. The developer implements a PMF generator class that extends the C# generator class. This class contains some PMF requirements: (1) the generated model component is a C# class that inherits the *Model* class, and (2) it contains the getter and setter methods of all model variables and parameters with the algorithm implemented in C#.

# 4. Discussion

The CyML language provides a relatively simple structure with few specifications that can express the algorithm of a biophysical process involved in crop growth and development. The real interest of this language is to provide a common method to describe a process with the capacity to be integrated automatically in various platforms. CyMLT provides export capabilities in many languages and platforms, enabling users to focus on the scientific aspect of their model rather than on the internal knowledge of platforms' specificities. A model component can be reused, improved, integrated and simulated in various platforms. This improves the diffusion of models, sharing them as a software and scientific artifacts, and thus, enhancing transparency and reproducibility of crop models. Moreover, with CyML, the model development may become a collaborative task of different groups of model builders with the possibility to compose different model units provided by different platforms.

For crop modelers, learning a new language with its own learning curve adds a level of complexity to an existing complex landscape of languages and tools. We designed CyML to minimize this added complexity by choosing a language that is very close to existing languages. The main source of complexity is in the model specification. The modeler has to specify the type of inputs and outputs, the documentation and unit tests. While this increases the complexity of the design of a new model, it provides an explicit and rigorous specification and enhances the transparency of the model and its reproducibility and reusability in different contexts. A transformation system embeds platform specificities to automatically generate model components conform to specific platforms. This makes the complexity of component integration in different platforms identical with a wide availability.

Several approaches and solutions exist to transform source code from one language to many higher-level programming languages (Baxter et *al.* 2004; Plaisted 2013; Schaub and Malloy 2016). They demonstrate the usefulness of source-to-source transformation systems in the development of reusable software libraries. For instance, De Paolis and Bourdot (2018) allow for the implementation of motion controllers of virtual humans, which are re-used in multiple game engines. Their system is based on Haxe, a language that offers the capability to transform Haxe code into many programming languages. However, like most available code transformation systems, the generated code depends on the transformation system. Likewise, Cython generates code into the C and C++ languages that have a high performance but the generated code has a low readability, therefore, making it difficult to understand and to maintain. To our knowledge, no solution exists to transform PBM algorithms in different languages considering the specificities of different modeling platforms. This transformation is useful in the sense that model components are not just code but embed scientific knowledge that should be preserved. In this work, we also propose a system that includes algorithm error checking with explicit error messages to guide developers. CyML addresses several issues encountered in current PBM frameworks, namely:

- reproducibility: a crop model or algorithm can be written once and automatically made available in different languages and platforms;

- reusability: a model can be reused and composed with other models of a specific platform;

- transparency: model algorithms are implemented using a common approach regardless of the crop simulation platform, and maintain the biophysical process knowledge.

Our approach and strategy should greatly reduce the implementation errors and improve model reproducibility. However, neither the definition of a language nor its transformation is approached without certain constraints, essentially due to the tradeoffs between generality and abstraction.

## 4.1 CyML transformation challenges

We provide a new language with a transformation system to produce code correctness. However, some inconsistencies or complexities could appear depending on the target language. First, the current version of CyML does not handle the type overflow. It means that errors related to overflow could not be detected at the CyML system level. For example, the generation of the Fibonacci recursive function in Python by just removing declaration types could lead to the crash of the system due to the Python recursion limit, whereas the generated code will not produce any error in Java but the result will rapidly overflow. A method to detect overflow can be implemented to avoid this type of error at run-time level. Moreover, CyML can be extended to support 64-bit C double type. Second, CyML provides primitive types whose equivalence in some platforms are objects with some properties. This means that coding an existing model algorithm in CyML could require an additional CyML external function to emulate the properties of these objects. Third, CyML has some limitations with data type conversion. For example, *Datetime type* is not supported in Fortran or C++. In this case, CyML converts it into strings. However, the translator could be extended to depend on specific libraries used by simulation platforms to perform the transformation. Finally, some platforms are close to the philosophy of their underlying language (e.g. DSSAT, BioMA, OpenAlea) whereas others extend their language with a high-level specificity (Record, Simplace) that requires a complex transformation.

## 4.2 Lower the barrier of crop simulation platforms

The main barrier to exchange and reuse of model components between simulation platforms is the specificities embedded in the algorithm implementation. CyML intends to lower the barrier of platform specificities. Our analysis of several platforms showed that each platform adopts a standard to implement model algorithms that does not vary from one implementation to another. The knowledge of platform requirements offers the possibility to integrate them into CyMLT in order to make their components available to many modeling platforms. We did not conduct a performance analysis but the cost of implementation is reduced by an order of magnitude compared to the time used to manually re-encode the same model into each platform without considering the inherent errors added during the process. CyML supports not only the transformation of the algorithm of unit models, but it also provides the evaluation of composite models by calling in sequential order models that are encapsulated into it. It also proposes a way to produce unit tests for each unit model algorithm in different languages based on the specifications of the inputs, outputs and parameter values. It checks the validity of the generated source code ensuring that all transformation results give the same results. It should be noted that CyML adds unit test functionality to platforms that do not use test-driven development.

## 4.3 CyML for model reuse and reproducibility

CyML implements PBM components with a functional and procedural approach. A component describing a biophysical process (e.g. phenology, soil water balance, photosynthesis) can be decomposed into independent components, which can be implemented and composed in CyML. Components implemented at a high granularity embed more scientific knowledge, but the component becomes less reusable. The implementation of a component into small functions (unit models) enhances its readability, reduces the distance between its expression as equations or mathematical expressions and its implementation, and reduces its maintenance cost. CyML is designed to tackle the reproducibility of PBM components. Although PBM are described in scientific publications and their code are increasingly publicly accessible, the reproducibility of the results remains a fundamental issue. Their implementation requires a procedural or functional language that is shared between simulation platforms to ensure their reproducibility. It is, therefore, useful to propose code in the language and that follow the specifications of the target platforms. The automatic transformation of model algorithms into different languages and simulation platforms is essential for interoperability and code reuse. CyML users can implement a model in CyML and transform the algorithms into various targets by using CyMLT. Hence, CyML aims at promoting PBM re-usability and interoperability through a transformation system that parses model specifications and knowledge needed to transform algorithms.

## 4.4 Scope of CyML language

CyML is a subset of the Cython language. Thus, it does not include many features found in general-purpose programming languages. This choice of language limitation has its strengths and weaknesses. The method presented herein differs from existing model interchange platforms in that it generates source code with different programming paradigms and it associates model specifications to algorithms to enhance code analysis. It allows a common implementation of the dynamics of biophysical processes by removing the specificities of the languages and platforms. It improves the readability of the code since the structure of the code and the characteristics of languages are shared by modeling platforms. It ensures the mapping of the abstract representation to other languages or platforms. Indeed, this language limitation reduces ambiguity in the language transformation since the base language (Cython) has some features that cannot be transformed into some target languages. With CyML, different processes provided by different platforms can be represented and composed regardless of the platforms, which enables to define a new white-box component reusable by other platforms. CyMLT provides a reuse approach that is opposite to a black-box approach where the composition of model components is bound to the execution platform targeted by its modules (Van Evert *et al.* 2005).

CyML does not interact with the simulation paradigms of the platforms. Its sole concern is to represent and transform the process models. Its evaluation capabilities are only used to check the correctness of the transformation. Moreover, CyML does not provide a formalism to link model components with data to build a modeling solution. Thus, the processes to read inputs, parameters values and write output values in a file is separated from the algorithm implementation given that it reduces reusability.

Although CyML focuses on the implementation and reuse of biophysical models, it could be used in general purpose. Thus, any code that can be implemented with CyML features can be transformed into different languages without associating specifications files.

## 4.5 Toward a standard language

The development of CyML and its transformation system addresses the need of the plant and crop modeling community to enhance research collaboration by improving the capacity to exchange and reuse PBM components. The theoretical interest to provide a common approach to implement model response has been demonstrated (Holzworth *et al.* 2014). However, despite the success of simulation platforms around which different communities are built, and some proposal of declarative language implementation, the lack of a shared standard limits model reusability. This issue limits the performance of the activity of PBM intercomparison and improvement. The availability of CyMLT through AMEI will allow building a large community around this system and can make CyML a standard language providing a means to seamlessly compare independent biophysical processes or promote alternatives approaches.

## 4.6 Future developments

Several modelers have expressed their interest to extend CyMLT with other languages used by the plant and crop modeling community. The use of a well-annotated ASG with model specifications provides an intuitive representation of the model algorithms. This abstraction set up various analysis of the source code by generating different source code based on the target language features, software design and code conventions. With this flexibility offered by the ASG, future work can explore the extension of CyMLT with other imperative programming languages such as Matlab, Julia, JavaScript or other modeling platforms that use imperative languages.

Reuse of legacy PBM model components without the need to encode them into CyML could reduce the investment in model exchange and could increase the interest of the platforms. Therefore, the next step would be to provide a transpiler that transforms legacy model components from various

languages and simulation platforms into CyML code automatically. Such a many-to-many transformer would provide a complete system of interoperability of languages and simulation platforms.

CyMLT aims to enable the exchange and reuse of components between modeling platforms, notably between PBM and functional-structural plant modelling (FSPM) platforms. While crop growth models simulate plant growth and development at the scale of the canopy ($m^2$) or average plant level, FSPMs are individual-based models at the scale of the organ. The exchange (sharing) of model components between PBM and FSPMs would allow an efficient coupling of these two modeling approaches to model crop species or variety mixtures by capturing spatial heterogeneities and quantifying plant traits involved in crop mixture performance (Gaudio *et al.* 2019). Another application is the use of FSPMs in a model-driven phenotyping approach, where plant structural traits are estimated by reverse engineering a FSPM (Liu *et al.* 2019) and are then used as crop model input parameters to simulate the behavior of genotypes in target agro-climatic scenarios. Currently, CyML only allows for the representation of processes as functions and does not consider the plant's structure. To extend CyML to the FSPM community will require to extend CyML language and CyMLT to support complex data structures such as 3-dimensional geometry and topology.

The convergence of our approach of model reuse and reproducibility approach with other collaborations, like the *Crops in Silico* collaboration (Marshall-Colon *et al.* 2017), would greatly accelerate the development of the next generation of PBMs. The *Crops in Silico* collaboration aims at integrating model frameworks to build a complete crop *in silico* from the level of the genes to the level of the field or ecosystem using a software package, Yggdrasil (Lang 2019). Yggdrasil connects PBMs across programming languages by running asynchronously models in parallel. It requires to write wrappers in the different languages to process the asynchronous messages to manage model inputs and outputs. CyMLT may interact with Yggdrasil (*i*) to make available model components into the languages supported by Yggdrasil with their wrappers, (*ii*) to produce efficient components source code in various languages in order to improve the performance of the simulation in Yggdrasil; and (*iii*) by validating each component with unit tests before their integration. The interaction between CyML and Yggdrasil could enhance the integration of PBMs across different languages and scales. A complementary approach to the one presented here was demonstrated for the automated transformation of input files of four agricultural models (Samourkasidis and Athanasiadis 2020) enabling the discovery and reuse of data across modelling solutions. Together with AMEI they could ensure that a complete model implementation and accompanied data can be transformed between modelling solutions.

## 5. Conclusions

In this study, we defined a minimal language based on the Cython language to implement biophysical processes involved in plant and crop growth and development. We designed a system that transforms CyML source code to many target languages and simulation platforms. The association of model specifications in XML-based format with the description of model algorithm based on CyML specifications allows to annotate each variable used in the algorithm. With this approach we can produce code with different programming paradigms including object-oriented approach and with different software designs. We showed that this language is sufficient to express biophysical processes and to transform them in different target languages and simulation platforms. We argue that the abstract language offers some trade-off between generality due to the convergence of the platforms and the complexity hidden in each platform. Crop modelers should have some programming skill to implement a model in CyML but no other skills are needed to produce automatically a model component source code in various languages and platforms. This reuse approach will help modelers to improve the reproducibility of their models and their reuse and should enhance research collaborations and model improvement and use.

## Code

The CyMLT source code are available publicly on Github at https://github.com/AgriculturalModelExchangeInitiative/PyCrop2ML. Full documentation for CyML and CYMLT can be found at https://pycrop2ml.readthedocs.io.

## Source of Funding

CM was supported through a PhD scholarship from the French National Research Agency under the Investments for the Future Program, referred as ANR-16-CONV-0004. CP was partially supported by the H2020 IPM Decision #817617. IA was partially supported by the European Union Horizon 2020 Research and Innovation program (Grant #810775, DRAGON). The work of CREA was carried out in the frame of the project AGRIDIGIT – Digital Agriculture, funded by the Italian Ministry of Agriculture

## Conflict of interest

The authors declare no conflict of interest.

## Acknowledgements

CM acknowledges the support of INRAE Divisions AgroEcoSystem and NUM. PM acknowledges the support of INRAE Division AgroEcoSystem.

## Supporting Information

The Following additional information are available on the online version of this article.

## Literature Cited


Akeret, J. *et al.* (2015) 'HOPE: A Python just-in-time compiler for astrophysical computations', *Astronomy and Computing*. Elsevier B.V., 10, pp. 1–8. doi: 10.1016/j.ascom.2014.12.001.

Akin, E. (2003) 'Object-Oriented Programming via Fortran 90/95', *Object-Oriented Programming via Fortran 90/95*. doi: 10.1017/cbo9780511530111.

Aslam, M. A. *et al.* (2017) 'Can growing degree days and photoperiod predict spring wheat phenology?', *Frontiers in Environmental Science*, 5(SEP), pp. 1–10. doi: 10.3389/fenvs.2017.00057.

Asseng, S. *et al.* (2013) 'Uncertainty in simulating wheat yields under climate change', *Nature Climate Change*, 3(9), pp. 827–832. doi: 10.1038/nclimate1916.

Athanasiadis, I. N. *et al.* (2011) 'Enriching environmental software model interfaces through ontology-based tools', *International Journal of Applied Systemic Studies*, 4(1–2), pp. 94–105. doi: 10.1504/IJASS.2011.042205.

Athanasiadis, I. N. and Villa, F. (2013) 'A roadmap to domain specific programming languages for environmental modeling: Key requirements and concepts', *DSM 2013 - Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling*, pp. 27–32. doi: 10.1145/2541928.2541934.

Ausbrooks, R. *et al.* (2003) 'Mathematical Markup Language ( MathML ) Version 2 . 0 ( Second Edition )', *Mathematical Markup Language Specification TA - Ausbrooks, Ron*, 0(October), pp. 0–385. Available at: http://www.w3.org/TR/MathML2/.

Autumn Cuellar *et al.* (2006) *CellML 1.1 Specification*. Available at: https://www.cellml.org/specifications/cellml_1.1/index_html (Accessed: 20 February 2018).

Baxter, I. D., Pidgeon, C. and Mehlich, M. (2004) 'DMS®: Program transformations for practical scalable software evolution', *Proceedings - International Conference on Software Engineering*, 26(May), pp. 625–634. doi: 10.1109/icse.2004.1317484.

Behnel, S. *et al.* (2011) 'Cython: The Best of Both Worlds', *Computing in Science & Engineering*, 13(2), pp. 31–39. doi: 10.1109/MCSE.2010.118.

Behnel, S., Bradshaw, R. and Seljebotn, S. (2000) 'Cython: The best of both worlds Stefan', *Rehab Management: The Interdisciplinary Journal of Rehabilitation*, 13(6), pp. 32–36.

Bergez, J. E. *et al.* (2016) 'A new plug-in under RECORD to link biophysical and decision models for crop management', *Agronomy for Sustainable Development*. Agronomy for Sustainable Development, 36(1), pp. 1–8. doi: 10.1007/s13593-016-0357-y.

Bindi, M. *et al.* (2015) 'Modelling climate change impacts on crop production for food security', *Climate Research*, 65(February 2014), pp. 3–5. doi: 10.3354/cr01342.

Brisson, N. *et al.* (1998) 'STICS: a generic model for the simulation of crops and their water and nitrogen balances. I. Theory and parameterization applied to wheat and corn', *Agronomie*, 18(5–6), pp. 311–346. doi: 10.1051/agro:19980501.

Brown, H. E. *et al.* (2014) 'Plant Modelling Framework: Software for building and running crop models on the APSIM platform', *Environmental Modelling and Software*. Elsevier Ltd, 62, pp. 385–398. doi: 10.1016/j.envsoft.2014.09.005.

Brown, H., Huth, N. and Holzworth, D. (2018) 'Crop model improvement in APSIM: Using wheat as a case study', *European Journal of Agronomy*. Elsevier, 100(February), pp. 141–150. doi: 10.1016/j.eja.2018.02.002.

Bysiek, M., Drozd, A. and Matsuoka, S. (2017) 'Migrating legacy fortran to python while retaining fortran-level performance through transpilation and type hints', *Proceedings of PyHPC 2016: 6th Workshop on Python for High-Performance and Scientific Computing - Held in conjunction with SC16: The International Conference for High Performance Computing, Networking, Storage and Analysis*, (November), pp. 9–18. doi: 10.1109/PyHPC.2016.006.

Cary, J. R. *et al.* (1997) 'Comparison of C++ and Fortran 90 for object-oriented scientific programming', *Computer Physics Communications*, 105(1), pp. 20–36. doi: 10.1016/S0010-4655(97)00043-X.

Donatelli, M. *et al.* (2010) 'A Component-Based Framework for Simulating Agricultural Production and Externalities', in *Environmental and Agricultural Modeling:* Dordrecht: Springer Netherlands, pp. 63–108. doi: 10.1007/978-90-481-3619-3_4.

Donatelli, M. *et al.* (2014) 'A generic framework for evaluating hybrid models by reuse and composition - A case study on soil temperature simulation', *Environmental Modelling and Software*. Elsevier Ltd, 62, pp. 478–486. doi: 10.1016/j.envsoft.2014.04.011.

Enders, A. *et al.* (2010) 'The IMPETUS Spatial Decision Support Systems', in *Impacts of Global Change on the Hydrological Cycle in West and Northwest Africa*. Berlin, Heidelberg: Springer Berlin

Heidelberg, pp. 360–393. doi: 10.1007/978-3-642-12957-5_11.

Van Evert, F. *et al.* (2005) 'Convergence in integrated modeling frameworks', *MODSIM05 - International Congress on Modelling and Simulation: Advances and Applications for Management and Decision Making, Proceedings*, pp. 745–750.

Fernique, P. and Pradal, C. (2017) 'AutoWIG: Automatic Generation of Python Bindings for C++ Libraries'. Available at: http://arxiv.org/abs/1705.11000.

Gamma, E. *et al.* (1995) 'Design Patterns: Elements of Reusable Object-Oriented Software', *Addison-Wesley*. doi: 10.1016/b978-012663315-3/50005-8.

Gaudio, N. *et al.* (2019) 'Current knowledge and future research opportunities for modeling annual crop mixtures. A review', *Agronomy for Sustainable Development*, 39(2), p. 20. doi: 10.1007/s13593-019-0562-6.

Gleeson, P. *et al.* (2010) 'NeuroML: A language for describing data driven models of neurons and networks with a high degree of biological detail', *PLoS Computational Biology*, 6(6), pp. 1–19. doi: 10.1371/journal.pcbi.1000815.

He, J. *et al.* (2012) 'Simulation of environmental and genotypic variations of final leaf number and anthesis date for wheat', *European Journal of Agronomy*. Elsevier B.V., 42, pp. 22–33. doi: 10.1016/j.eja.2011.11.002.

Holzworth, D. *et al.* (2018) 'APSIM Next Generation: Overcoming challenges in modernising a farming systems model', *Environmental Modelling & Software*. Elsevier Ltd, 103, pp. 43–51. doi: 10.1016/j.envsoft.2018.02.002.

Holzworth, D. P., Snow, V., *et al.* (2014) 'Agricultural production systems modelling and software: Current status and future prospects', *Environmental Modelling and Software*. Elsevier Ltd, 72, pp. 276–286. doi: 10.1016/j.envsoft.2014.12.013.

Holzworth, D. P., Huth, N. I., *et al.* (2014) 'APSIM - Evolution towards a new generation of agricultural systems simulation', *Environmental Modelling and Software*. Elsevier Ltd, 62, pp. 327–350. doi: 10.1016/j.envsoft.2014.07.009.

Hoogenboom, G. *et al.* (2019) 'The DSSAT crop modeling ecosystem', in, pp. 173–216. doi: 10.19103/AS.2019.0061.10.

Hucka, M. *et al.* (2015) 'Promoting Coordinated Development of Community-Based Information Standards for Modeling in Biology: The COMBINE Initiative', *Frontiers in Bioengineering and Biotechnology*, 3(February), pp. 1–6. doi: 10.3389/fbioe.2015.00019.

Jones, J. W. *et al.* (2003) *The DSSAT cropping system model*, *European Journal of Agronomy*. doi: 10.1016/S1161-0301(02)00107-7.

Jones, J. W. *et al.* (2017) 'Brief history of agricultural systems modeling', *Agricultural Systems*. Elsevier B.V., 155(June), pp. 240–254. doi: 10.1016/j.agsy.2016.05.014.

Kluyver, T. *et al.* (2016) 'Jupyter Notebooks—a publishing format for reproducible computational workflows', *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pp. 87–90. doi: 10.3233/978-1-61499-649-1-87.

van Kraalingen, D. W. G. *et al.* (2020) 'WISS a Java Continuous Simulation Framework for Agro-Ecological Modelling', in, pp. 242–248. doi: 10.1007/978-3-030-39815-6_23.

Lang, M. (2019) 'yggdrasil: a Python package for integrating computational models across languages and scales', *in silico Plants*, 1(1). doi: 10.1093/insilicoplants/diz001.

Linge, S. and Langtangen, H. P. (2016) *Programming for Computations - Python*. Cham: Springer International Publishing (Texts in Computational Science and Engineering). doi: 10.1007/978-3-319-32428-9.

Liu, S. *et al.* (2019) 'Estimation of plant and canopy architectural traits using the digital plant phenotyping platform1[OPEN]', *Plant Physiology*, 181(3). doi: 10.1104/pp.19.00554.

Manceau, L. and Martre, P. (2018) 'SiriusQuality-BioMa-Phenology-Component'. doi: 10.5281/ZENODO.2478791.

Marshall-Colon, A. *et al.* (2017) 'Crops in silico: Generating virtual crops using an integrative and multi-scale modeling platform', *Frontiers in Plant Science*, 8(May), pp. 1–7. doi: 10.3389/fpls.2017.00786.

Martre, P. *et al.* (2006) 'Modelling protein content and composition in relation to crop nitrogen dynamics for wheat', *European Journal of Agronomy*, 25(2), pp. 138–154. doi: 10.1016/j.eja.2006.04.007.

Martre, P. *et al.* (2018) 'The agricultural model exchange initiative', in IICA (ed.) *7th AgMIP Global*

*Workshop*. San José, Costa Rica.

Midingoyi, C. A. *et al.* (2020) 'Crop2ML: The centralized framework for crop model component exchange and reuse'. doi: 10.5281/ZENODO.3911713.

Misse-chanabier, P., Aranega, V. and Polito, G. (2019) 'Illicium A modular transpilation toolchain from Pharo to C', (Vm).

Muller, B. and Martre, P. (2019) 'Plant and crop simulation models: powerful tools to link physiology, genetics, and phenomics', *Journal of Experimental Botany*, 70(9), pp. 2339–2344. doi: 10.1093/jxb/erz175.

Palosuo, T. *et al.* (2011) 'Simulation of winter wheat yield and its variability in different climates of Europe: A comparison of eight crop growth models', *European Journal of Agronomy*, 35(3), pp. 103–114. doi: 10.1016/j.eja.2011.05.001.

De Paolis, L. T. and Bourdot, P. (2018) 'Write-once, transpile-everywhere: re-using motion controllers of virtual humans across multiple game engines', (July), pp. E1–E1. doi: 10.1007/978-3-319-95282-6_51.

Plaisted, D. A. (2013) 'Source-to-Source Translation and Software Engineering', *Journal of Software Engineering and Applications*, 06(04), pp. 30–40. doi: 10.4236/jsea.2013.64A005.

Pradal, C. *et al.* (2008) 'OpenAlea: A visual programming and component-based software platform for plant modelling', *Functional Plant Biology*, 35(10), pp. 751–760. doi: 10.1071/FP08084.

Pradal, C. *et al.* (2015) 'OpenAlea : Scientific Workflows Combining Data Analysis and Simulation To cite this version : HAL Id : hal-01166298 OpenAlea : Scientific Workflows Combining Data Analysis and Simulation'.

Quinlan, D. and Liao, C. (2011) 'The ROSE Source-to-Source Compiler Infrastructure', *International Journal*, pp. 1–3.

R Core Team (2017) 'R: A Language and Environment for Statistical Computing'. Vienna, Austria. Available at: https://www.r-project.org/.

Rosenzweig, C. *et al.* (2013) 'The Agricultural Model Intercomparison and Improvement Project (AgMIP): Protocols and pilot studies', *Agricultural and Forest Meteorology*. Elsevier B.V., 170, pp. 166–182. doi: 10.1016/j.agrformet.2012.09.011.

Rötter, R. P. *et al.* (2011) 'Crop-climate models need an overhaul', *Nature Climate Change*. Nature Publishing Group, 1(4), pp. 175–177. doi: 10.1038/nclimate1152.

Samourkasidis, A. and Athanasiadis, I. N. (2020) 'A semantic approach for timeseries data fusion', *Computers and Electronics in Agriculture*. Elsevier, 169(December 2019), p. 105171. doi: 10.1016/j.compag.2019.105171.

Schaub, S. and Malloy, B. A. (2016) 'The design and evaluation of an interoperable translation system for object-oriented software reuse', *Journal of Object Technology*, 15(4), pp. 1–33. doi: 10.5381/jot.2016.15.4.a1.

Schilstra, M. J. *et al.* (2006) 'CellML2SBML: Conversion of CellML into SBML', *Bioinformatics*, 22(8), pp. 1018–1020. doi: 10.1093/bioinformatics/btl047.

Villa, F. (2001) 'Integrating modelling architecture: A declarative framework for multi-paradigm, multi-scale ecological modelling', *Ecological Modelling*, 137(1), pp. 23–42. doi: 10.1016/S0304-3800(00)00422-1.

Villa, F. *et al.* (2017) 'Semantics for interoperability of distributed data and models: Foundations for better-connected information', *F1000Research*, 6(2), p. 686. doi: 10.12688/f1000research.11638.1.

de Wit, A. *et al.* (2019) '25 years of the WOFOST cropping systems model', *Agricultural Systems*. Elsevier, 168(October 2017), pp. 154–167. doi: 10.1016/j.agsy.2018.06.018.

Zhao, C. *et al.* (2019) 'A SIMPLE crop model', *European Journal of Agronomy*. Elsevier, 104, pp. 97–106. doi: 10.1016/J.EJA.2019.01.009.