Semantic and modular representation of crop models using a declarative metalanguage.

Cyrille Ahmed Midingoyi

To cite this version:

Cyrille Ahmed Midingoyi. Semantic and modular representation of crop models using a declarative metalanguage.. Modeling and Simulation. Montpellier SupAgro, 2020. English. NNT : . tel-03357052

HAL Id: tel-03357052
https://hal.science/tel-03357052

Submitted on 28 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE MONTPELLIER SUPAGRO

En Agronomie

École doctorale GAIA – Biodiversité, Agriculture, Alimentation, Environnement, Terre, Eau
Portée par

Unité de recherche Laboratoire d'Ecophysiologie des Plantes sous Stress Environnementaux

Représentation sémantique et modulaire des modèles de culture à l'aide d'un métalangage déclaratif

Présentée par Cyrille Ahmed MIDINGOYI
Le 18 décembre 2020

Sous la direction de Pierre MARTRE
et Frédérick GARCIA

Devant le jury composé de

Marianne HUCHARD, Professeur, Université de Montpellier, HDR	Présidente du jury
Eric RAMAT, Professeur, Université du Littoral Côte d'Opale, HDR	Rapporteur
Gerhard BUCK-SORLIN, Professeur, AgroCampus Ouest, HDR	Rapporteur
Gaëtan LOUARN, Chargé de recherche, INRAE	Examinateur
Pierre MARTRE, Directeur de recherche, INRAE, HDR	Directeur
Frédérick GARCIA, Directeur de recherche, INRAE, HDR	Co-Directeur (Invité)
Christophe PRADAL, Chargé de recherche, CIRAD	Co-encadrant
Myriam ADAM, Chargé de recherche, CIRAD	Invitée

l'institut Agro
agriculture • alimentation • environnement

Remerciements

J’adresse toute ma profonde gratitude à tous ceux qui ont œuvré directement ou indirectement à la réalisation de ce travail.

Je souhaite tout d’abord remercier chaleureusement :

Pierre MARTRE, mon directeur de thèse, directeur du laboratoire LEPSE, de m’avoir donné l’opportunité de prouver mes capacités de conduire cette thèse. En dépit de son agenda chargé, il a toujours su se rendre disponible au moment opportun pour assurer le bon déroulement des travaux. La confiance qu’il a placée en moi est l’une de mes sources de motivation. Il a élargi mon champ de connaissances en m’intégrant dans la communauté de modélisateurs de culture à travers l’initiative d’échange des modèles en agriculture (AMEI). Je saisis cette occasion pour adresser toute ma reconnaissance aux membres de l’initiative.

Christophe PRADAL, mon co-encadrant de thèse. Il m’est difficile d’exprimer en quelques phrases ma reconnaissance envers lui pour l’aboutissement de ce travail. Sa patience, sa disponibilité et ses conseils m’ont permis de vivre une expérience stimulante, enrichie par son expertise et sa rigueur scientifique qui, sans doute, impacteront ma carrière professionnelle. Dire qu’il n’a été qu’un encadrant serait un euphémisme. Merci pour ces moments sympathiques en compagnie de ton adorable famille.

Frédérick GARCIA, mon co-directeur de thèse pour ses précieux conseils, suggestions et encouragements tout au long de ces trois années.

Mes vifs remerciements aux rapporteurs, pour l’intérêt qu’ils ont porté à cette thèse et à tous les membres du jury, pour avoir accepté d’examiner mon travail. Merci aux membres de comité de thèse, dont Eric Justes, le référent de l’École Doctorale GAIA dans le cadre de ma thèse, pour leur regard scientifique et pour leurs recommandations ayant conduit à l’achèvement de cette thèse.

Je tiens également à remercier tout le personnel du laboratoire LEPSE, notamment l’ex-directeur de l’unité, Bertrand Muller, Marie-Françoise, Anais, Marie-Claude, pour l'accueil chaleureux et les conditions de travail qui m’ont été offertes. Je n’oublierai pas mes collègues, en particulier Sibylle, Anh, Stéphane et Adriann.

J’exprime toute ma reconnaissance à Seydou Traore, Alhassane Agali, experts du Centre Régional AGRHYMET et Christian Baron, chercheur au CIRAD pour leur encouragement, la confiance qu’ils ont toujours portée en moi et leur implication à différents niveaux pour mon arrivée à Montpellier.

Merci aux compatriotes béninois résidents à Montpellier en particulier Japhet, Kola, Nawalyath, et la famille Kuissode pour leur sollicitude et les moments copieux que nous avons partagés ensemble. Je sais tout particulièrement gré à Lorène pour son accompagnement et son soutien tout au long de ces années.

Je suis redevable à mon père, ma mère, mes frères et sœurs pour leur soutien moral et spirituel, et leur confiance.

Enfin je remercie mon épouse Amanda pour son soutien quotidien indéfectible, pour sa patience, son sacrifice durant cette longue absence et pour son enthousiasme à voir le bout de ce long périple soldé avec succès. Merci d'être endurante avec mes adorables enfants, Noah, Faith et Rayane, pour qui cette thèse doit être un socle de l'espérance et du dévouement. Je m'excuse de n'avoir pas été à vos côtés durant ces longues années. Puisse Dieu nous accorder les bénéfices de ce noble sacrifice. Amen

List of Abbreviations

APSIM: Agricultural Production Systems sIMulator

ATL: ATLAS Transformation Language

BioMA: Biophysical Model Applications

Crop2ML: Crop Meta Modeling Language

CyML: Cython-derived Modeling Language

DEVS: Discrete Event System Specification

DSSAT: Decision Support System for Agrotechnology Transfer

DSL: Domain Specific Language

DSML: Domain-Specific Modeling Language

GUI: Graphical User Interface

IDE: Integrated Development Environment

MDA: Model Driven Architecture

MDE: Model Driven Engineering

PBM: Process-Based Model

PMF: Plant Modeling Framework

QVT: Query View Transformation

RECORD : REnovation et COoRDination de la modélisation des cultures pour la gestion des agro-écosystèmes

SBML: System Biology Markup Language

SIMPLACE: Scientific Impact assessment and Modelling Platform for Advanced Crop and Ecosystem management

SysML: Systems Modeling Language

UML: Unified Modeling Language

Glossary

Abstract class: a set of operations which all objects that implement the protocol must support

Algorithm: a set of coherent equations, mathematical expressions with or without logical rules to solve a specific problem

Component: a part of a system; a piece of software representing plant and/or soil processes that is used to compose a crop model (e.g. crop, light interception, water uptake, soil water, or soil C and N components)

Conceptual model: a concise and precise consolidation of all goal-relevant structural and behavioral features of the system presented in a predefined format. It provides foundation for the development of the simulation program.

Design pattern: a general reusable solution to a commonly occurring problem in software design.

Modularity: the property of a system to be made up of relatively independent, but interlocking components or parts.

Reverse engineering: a process of analyzing a subject system in order to identify the system's constituents and create representations in other forms or at higher levels of abstraction

Strategy design pattern: a software design that defines a family of algorithms, encapsulates each one, and makes them interchangeable

Wrapper: a class that serves to mediate access to another.

Source of Funding

This work 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 and by INRAE AgroEcosystem and MathNum divisons.

Résumé

L’hétérogénéité des plateformes de modélisation de culture en matière de langage d'implémentation, de motif de conception et de contraintes d’architecture logicielle limite la réutilisation des composants de modèles en dehors de la plateforme dans laquelle ils ont été développés. Notre objectif est de proposer une approche de réutilisation basée sur une forte abstraction des composants de modèle. Pour ce faire, nous avons identifié des concepts qui ont permis de définir un métalangage de spécification des composants et un langage métier minimal de description des algorithmes indépendamment des plateformes. Un système de transformation basée sur ces concepts a permis de générer de façon transparente et automatique des composants compatibles à différentes plateformes de modélisation. Dans cette thèse, nous avons montré que la description unifiée des composants de modèle avec des concepts partagés permet de lever les contraintes des plateformes et favorise la réutilisabilité des composants de façon transparente.

Abstract

The heterogeneity of crop modeling platforms in terms of implementation language, design pattern, and software architecture constraints, limits the reuse of model components outside the platform in which they have been developed. Our objective is to propose a reuse approach based on a high level of abstraction of model components. To this end, we have identified some concepts that made it possible to define a component specification language and a minimal domain language for the description of algorithms regardless of platform specificities. A transformation system based on these concepts allowed us to generate seamlessly platform-compliant components. We have shown that a unified description of model components with shared concepts lift constraints of platforms and increases reusability of components.

Résumé étendu

1. Contexte

1.1. Motivation

Depuis plusieurs décennies, les modèles de culture font l'objet d'une attention particulière. Ils fournissent une représentation conceptuelle du continuum sol-plante-atmosphère et permettent de décrire la croissance et le développement des cultures en interaction avec leur milieu. Ils sont de plus en plus développés et étendus pour répondre à un large éventail d'applications. Les avancées en génie logiciel constituent l'un des facteurs significatifs du développement accru des modèles de culture en favorisant différentes approches d'implémentation. Elles ont permis aux modélisateurs de concevoir des modèles génériques basés sur des fonctions physiologiques communes impliquées dans la croissance et le développement d'une grande variété de cultures (par exemple, AquaCrop (Steduto et al., 2009), SPASS (Wang & Engel, 2000), STICS (Brisson et al., 1998)) ou d'élaborer des plateformes de modélisation facilitant le développement et la réutilisation des modèles. On peut citer, entre autres, les plateformes RECORD (J. E. Bergez et al., 2013), BioMA (Donatelli et al., 2012), OpenAlea (Pradal *et al.*, 2015), SIMPLACE (Gaiser et al., 2013), APSIM (H. E. Brown et al., 2014; Holzworth et al., 2018), DSSAT (Hoogenboom et al., 2019), ou CROSPAL (Adam *et al.*, 2010a). Cette diversité est aussi liée aux progrès de l'agriculture numérique qui offre une masse de données produites par des capteurs à courte portée ou à distance permettant d'alimenter les modèles dont les sorties peuvent soulever de nouvelles interrogations, amenant à remettre en cause le formalisme des modèles, notamment les mécanismes causaux des réponses de la plante à son environnement.

La diversité des modèles a rapidement amené la communauté des modélisateurs de culture à comparer la performance des modèles en vue de les améliorer en agrégeant leurs connaissances ou en introduisant d'autres innovations fournies par divers groupes de recherche sous l'égide de différents projets de collaboration. Les projets de recherche menés dans le cadre d'inter-comparaisons de modèles (Palosuo et *al.* 2011; Rötter et *al.* 2011; Asseng et *al.* 2013; Aslam et *al.* 2017) ont permis de mettre en évidence les différences entre les sorties des modèles sans pouvoir déterminer les sources d'incertitude ni analyser les processus qui y sont impliqués (Muller & Martre, 2019). Ces résultats d'inter-comparaison de modèles montrent le potentiel et les limites des modèles et mettent en évidence la nécessité de tester les modèles au niveau des processus, mais aussi d'échanger entre modélisateurs et plateformes de modélisation les composants des modèles. L'échange de composants de modèles s'est avéré nécessaire pour permettre d'analyser diverses hypothèses de modélisation et pour améliorer la robustesse des modèles.

1.2. Problématique

Malgré leur intérêt et leurs avancées, la diversité des plateformes de modélisation en matière de langages de programmation, de motifs de conception, de contraintes d'architecture logicielle a eu un impact négatif sur le progrès de la modélisation des cultures. Elle a entraîné une perte de transparence pour les modélisateurs et un ralentissement du développement de nouveaux formalismes en raison d'un manque de standards évolutifs dans la mise en œuvre et la réutilisation des modèles. Les composants de modèles ne sont pas réutilisables en dehors de la plateforme dans laquelle ils ont été développés, et peu d'avantages sont tirés des composants de modèles existants développés par d'autres plateformes. Par ailleurs, bien que l'intérêt de modularité soit reconnu depuis longtemps (Donatelli et al., 2014; Timlin et al., 1996), la modularité est explicitement abordée qu'au niveau de la phase d'implémentation des composants. De même, les hypothèses et descriptions (méta-données) sont rarement accessibles dans le code source des composants (Athanasiadis & Villa, 2013). La réutilisation des composants nécessite la connaissance des objectifs de modélisation et un haut niveau d'abstraction. Actuellement, la majorité des plateformes ne représentent pas explicitement le modèle conceptuel des composants qui est un artefact réutilisable. La représentation conceptuelle reste souvent informelle ou dans l'esprit. En outre, la publication de composants de modèle dans les revues scientifiques ne fournit pas une description suffisante des processus modélisés (Keller & Dungan, 1999) pour permettre de reproduire et juger de la fiabilité des résultats scientifiques fortement liés à la plateforme dans laquelle le composant a été mis en œuvre et testé (Cohen-Boulakia et al., 2017; Hinsen, 2016).

Etant donné que les modèles ou les composants de modèles sont souvent directement représentés sous forme de programmes informatiques, les approches de réutilisation consistent souvent à soit les utiliser comme une boîte noire (Rizzoli *et al.*, 2008) ou à les traduire manuellement pour les adapter aux exigences de la plateforme cible. L'approche de boîte noire est intéressante pour gérer l'hétérogénéité des composants à travers l'encapsulation de composants dans une nouvelle plateforme mais elle nécessite la mise en œuvre d'algorithmes complexes, réduit la connaissance du comportement interne du composant et ne permet pas l'extension ou la rénovation des composants. D'un point de vue social, les composants sous la forme de boîtes noires sont considérés peu fiables (Janssen et al., 2017). Le recodage d'un composant nécessite des compétences en programmation dans le langage dans lequel le composant est implémenté. Ainsi, l'absence d'un système de transformation automatique peut augmenter le temps et le coût de développement. Un tel système est nécessaire pour garantir la cohérence entre les modèles de simulation (code source) et les modèles conceptuels.

1.3. Objectif et questions de recherche

L'objectif principal de cette recherche est de définir une approche de réutilisation de composants de modèles de culture entre les plateformes de modélisation et de simulation. Elle sera centrée autour de la définition de concepts partagés permettant de représenter uniformément les composants de modèles (pour répondre à la question de la conceptualisation des composants) et sur laquelle vont s'appuyer des transformations automatiques entre les plateformes. Pour aborder cette problématique de réutilisation des composants de modèles et atteindre notre objectif de recherche, deux questions principales sont étudiées :

- Existe-t-il une représentation commune des composants de modèles partagée entre les plateformes de modélisation et de simulation des cultures ?
- Comment pouvons-nous concevoir un système de transformation pour atteindre l'objectif de recherche ?

2. Principaux résultats

2.1. Proposer un ensemble de concepts pour une représentation partagée des composants des modèles entre les plateformes de modélisation

Nous abordons le manque de représentation conceptuelle ou de description explicite des composants de modèle et la prise en compte de la modularité dans les plateformes de modélisation. Nous supposons qu'une approche de réutilisation transparente passe par la définition d'un ensemble de concepts partagés. Pour cela, nous avons proposé un ensemble de concepts (Fig. 1) qui permettent de représenter explicitement et uniformément les composants. Ainsi, notre approche de réutilisation est centrée sur ces concepts. Sur cette base, nous avons défini un métalangage déclaratif Crop2ML permettant de décrire la spécification des modèles unitaires et leur composition pour répondre à la question de la modularité. Un modèle unitaire est constitué d'un ensemble d'éléments tels que sa description, une liste d'entrées, une liste de sorties, l'initialisation des variables d'état, une liste de fonctions mathématiques et un lien où est décrit l'algorithme du modèle. Il inclut également une représentation unifiée des tests unitaires avec des jeux de paramètres. Ses éléments structurés et leurs attributs constituent la grammaire du métalangage et permettent de vérifier si un modèle est conforme à Crop2ML. Un modèle composite est un graphe de modèles connectant les sorties d'un modèle aux entrées d'un autre. Ces concepts permettent donc de représenter les composants indépendamment des spécificités des plateformes de modélisation. En proposant une définition de tests unitaires pour chaque modèle unitaire avant que le processus ne soit implémenté, le développeur de modèles est obligé d'écrire

des unités qui sont faiblement couplées à partir d'autres unités (Holzworth et al., 2011), facilitant ainsi la réutilisation.

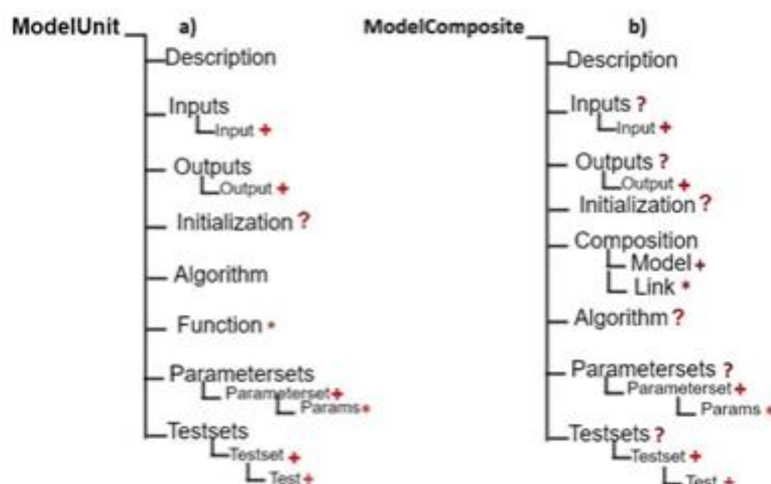

Figure 1 : Concepts Crop2ML pour la spécification de modèle model. (a) Model unitaire. (b) Model composite. “+” : 1 ou plusieurs ; “*” : zéro ou plusieurs ; “?” : zéro ou un.

2.2. Définir un langage commun de représentation de la dynamique des processus biophysiques dans les modèles de culture (CyML)

Etant donné que les algorithmes des composants sont décrits non seulement par des équations aux différences finies mais aussi par un ensemble d’expressions mathématiques avec des structures de contrôle, nous avons fait une analyse des langages utilisés par les plateformes de modélisation. Cette analyse consiste à identifier les différentes constructions nécessaires et suffisantes pour la description des algorithmes sans la prise en compte des spécificités des plateformes. L’idée est d’aboutir à une description qui soit la plus proche de la représentation mathématique des composants. Nous avons donc défini un langage minimal de haut niveau à partir des constructions identifiées compatibles entre les langages des plateformes (Fig. 2). Ce langage CyML est une restriction de Cython et permet de représenter l’algorithme des composants.

2.3. Définir un système de transformation de Crop2ML vers les plateformes de modélisation

La principale approche souvent utilisée pour aborder la réutilisation des composants de modèles est de traduire manuellement le modèle ou de l'encapsuler pour répondre aux exigences des plateformes cibles. Si la première approche est lourde et exige des compétences dans différents langages et la connaissance des spécificités des plateformes, la seconde entrave la connaissance derrière les composants et l'utilise comme une boîte noire. Nous abordons la question de la réutilisation avec une approche boîte blanche. Nous avons construit un système de transformation qui permet de traduire automatiquement un modèle Crop2ML dans de nombreux langages et plateformes de modélisation de culture.

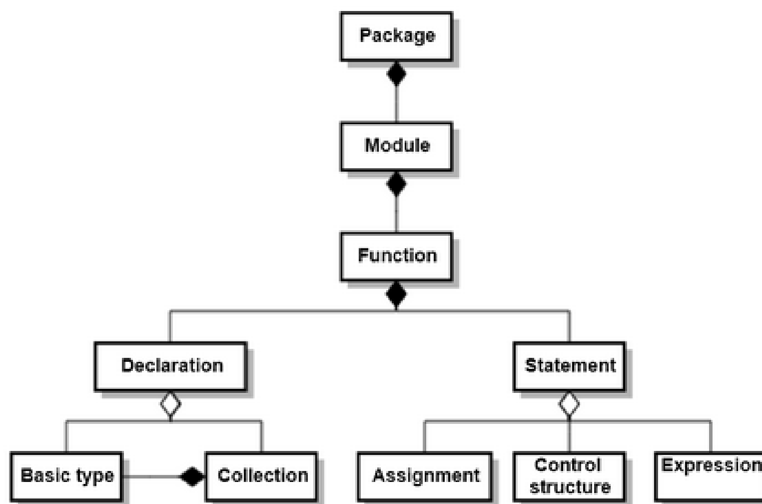

Figure 2 : Constructions du langage CyML

Il consiste à transformer le modèle Crop2ML associé aux algorithmes en un arbre syntaxique abstrait puis en une représentation uniforme indépendante de tout langage nommée ASG. Un ASG est un arbre sémantique abstrait obtenu à partir de la réécriture d'un AST. Cet arbre permet d'associer les informations sur les variables du modèle de spécification aux variables de l'algorithme en vue d'analyser la cohérence entre l'algorithme et le modèle de spécification. Enfin un générateur de code permet de convertir l'ASG en codes source dans différents langages et en composants compatibles aux exigences des plateformes. Cette transformation permet également de générer la documentation des composants compatible et liée aux spécifications des composants.

2.4. Proposer une approche pour inférer le modèle conceptuel à partir des différentes implémentations dans les plateformes de modélisation

Nous avons ensuite défini un ensemble de principes qui permet d'inférer le modèle Crop2ML à partir des composants de modèles de plateformes. Ces principes consistent en la formalisation de patterns permettant d'identifier les éléments de spécification des composants de modèles en vue de les traduire vers Crop2ML. Cette inférence associée à CyMLT permet d'aboutir à un système de transformation complet, voire d'interopérabilité entre les plateformes de modélisation.

2.5. Tester l'applicabilité de notre approche de réutilisation

Pour tester l'applicabilité de notre approche de réutilisation, nous avons implémenté un environnement multilangage d'échange et de réutilisation des composants de modèle de culture entre les plateformes de modélisation et de simulation. Cet environnement comporte plusieurs phases, dont la création, l'édition, la composition, la vérification, la validation, la transformation, la documentation et la visualisation des modèles. Un prototype de cet environnement a été implémenté à partir de JupyterLab, un environnement interactif de développement Web.

3. Conclusion et Perspectives

Dans cette thèse, nous avons abordé le problème de la réutilisation des composants des modèles entre les plateformes de modélisation et de simulation des cultures. Ainsi nous avons proposé une architecture de réutilisation de composants centrée sur la définition d'un ensemble de concepts permettant d'aboutir à une représentation unifiée et partagée de composants entre les plateformes de modélisation. Ces concepts ont permis de définir un langage de spécification des modèles Crop2ML puis un langage métier minimal pour la description des algorithmes des modèles CyML. Cela permet ainsi de séparer la conception des modèles de leur mise en œuvre, cachant ainsi les détails de l'implémentation. L'implémentation du modèle est dérivée de sa spécification à travers un système de transformation extensible capable de générer des composants compatibles avec les plateformes de modélisation, dont DSSAT, BioMA, SIMPLACE, Record, et OpenAlea. En vue de permettre aux plateformes de garder tous leurs avantages et d'aboutir à une interopérabilité entre les plateformes, nous avons aussi mis en place des stratégies pouvant permettre d'inférer le modèle conceptuel à partir des composants des plateformes. Une représentation de haut niveau est une base pour mieux comprendre les hypothèses sous-jacentes et faciliter la collaboration entre les groupes de modélisation. Un certain nombre d'orientations futures ont été identifiées au cours de nos travaux.

3.1. Améliorations de Crop2ML

Premièrement, nous n'avons défini aucune notion de variables composites ou de structure complexe de données. Une variable composite est une variable composée de deux ou plusieurs variables ou mesures qui sont fortement liées entre elles (Ley, 1972). L'utilisation de variables composites est une pratique courante dans le développement des modèles. Les variables individuelles qui composent une variable composite peuvent être des variables catégorielles (par exemple, les stades de développement) ou des cohortes d'organes. Pour convertir en Crop2ML un composant existant ayant des variables composites, nous décomposons d'abord manuellement la variable composite en plusieurs variables individuelles selon les structures de données de Crop2ML. Cela réduit l'automatisation du système de transformation.

Deuxièmement, nous n'avons pas une approche pouvant aider à sélectionner judicieusement les composants de modèle en fonction des connaissances biophysiques (Adam *et al.*, 2010b). Actuellement, la spécification du modèle est la seule source fournissant le contexte de modélisation par la provenance du composant et sa description. Cependant, nous n'avons pas de concept qui permet d'assurer la composition de contextes (R. Lara et al., 2006) ou qui guide l'utilisateur pour une composition sémantique des composants de modèles. Il est donc utile d'intégrer dans Crop2ML une approche de sélection des composants basée sur des connaissances approfondies qui pourra aider à construire des composants compatibles avec les exigences scientifiques des modèles.

3.1.1. Vers une plateforme multi-échelle ?

Une plateforme pour la connexion des plateformes de modélisation PBM et FSPM

Crop2ML vise à permettre l'échange et la réutilisation de composants entre les plateformes de modélisation, notamment entre les plateformes de croissance des cultures et de modélisation fonctionnelle et structurelle des plantes (FSPM). Alors que les modèles de croissance des cultures simulent la croissance et le développement des plantes à l'échelle du couvert végétal (m^2) ou au niveau d'une plante moyenne, les FSPM sont des modèles basés sur la plante individuelle ou à l'échelle de l'organe. L'échange (partage) de composants de modèles entre des modèles de croissance des cultures et des FSPM permettrait un couplage efficace de ces deux approches de modélisation pour modéliser les mélanges de variétés ou d'espèces en capturant les hétérogénéités spatiales et en quantifiant les caractéristiques des plantes impliquées dans la performance de ces mélanges (Gaudio *et al.* 2019). Une autre application est l'utilisation des FSPM dans une approche de phénotypage piloté par modèle, où les traits structurels des plantes sont estimés par rétro-ingénierie d'un FSPM (Liu *et al.* 2019) et sont ensuite utilisés comme paramètres d'entrée des modèles de culture pour simuler le comportement des génotypes dans des scénarios agroclimatiques cibles. Actuellement, Crop2ML permet uniquement de représenter

les processus sous forme de fonctions et ne prend pas en compte la structure de la plante. Pour étendre Crop2ML à la communauté FSPM, il faudra supporter des structures de données complexes telles que la géométrie et la topologie 3D.

Un lien entre Crop2ML et les plateformes de modélisation intégrative

La convergence de notre approche de réutilisation et de reproductibilité des composants de modèles avec d'autres initiatives de réutilisation, comme *Crops in silico* (Marshall-Colon et al., 2017) accélérerait considérablement le développement de la prochaine génération de PBM. L'initiative *Crops in Silico* vise à intégrer des plateformes de modélisation pour construire une culture *in silico* complète du niveau des gènes au niveau de la parcelle ou de l'écosystème en utilisant un logiciel, Yggdrasil (Lang, 2019). Yggdrasil permet de connecter des modèles hétérogènes en les exécutant de façon asynchrone ou en parallèle. Cela nécessite l'encapsulation des modèles dans différents langages pour traiter les messages asynchrones afin de gérer les entrées et les sorties des modèles. Crop2ML peut interagir avec Yggdrasil (*i*) pour mettre à disposition des composants de modèle dans les langages supportés par Yggdrasil, (*ii*) pour produire du code source de composants efficaces dans différents langages afin d'améliorer les performances de Yggdrasil ; et (*iii*) en validant chaque composant avec des tests unitaires avant leur intégration. L'interaction entre CyML et Yggdrasil pourrait améliorer l'intégration des PBM dans différents langages et à différentes échelles.

3.1.2. Plateforme d'inter-comparaison des solutions de modélisation

Une perspective consiste à comparer les modèles de simulation fournis par les plateformes de modélisation et de simulation des cultures. Il faut pour cela proposer un modèle de calcul générique pour assurer l'ordonnancement des modèles Crop2ML. Malgré les différences entre les plateformes de modélisation et de simulation des cultures, certaines caractéristiques communes ont été identifiées. Celle-ci ont permis de représenter les processus biophysiques indépendamment de leurs spécificités. Nous avons développé Crop2ML en partant du principe que les différences entre les sorties des modèles sont dues aux approches de modélisation (algorithmes) dans les processus individuels. Cependant, les différences dans les modèles de calcul (modèle séquentiel [par exemple, BioMA, Simplace, DSSAT], flux de données [OpenAlea], événement discret [Record]) pourraient également avoir un fort impact sur les résultats de simulation, mais nous n'en tenons pas compte dans notre thèse. Crop2ML pourra ainsi être étendu pour prendre en charge différents modèles de calcul. Une approche complémentaire à notre système de transformation présenté a été démontrée pour la transformation automatisée des fichiers d'entrée de quatre modèles de culture (Samourkasidis et al., 2019) Cette approche permet de découvrir et réutiliser des données à travers des solutions de modélisation. La combinaison de cette approche avec Crop2ML pourrait conduire à une implémentation complète de modèles en lien avec les données

associées, ce qui permettrait de quantifier les processus des modèles de culture de manière robuste et répétable.

3.1.3. Extension de Crop2ML avec l'annotation sémantique des modèles

La transformation de Crop2ML vers les plateformes est bien réalisée puisque le système de transformation est conçu pour prendre en charge les spécificités des plateformes cibles. Cependant, la sémantique de modèles Crop2ML repose essentiellement sur les concepts communs définis pour décrire à un haut niveau d'abstraction les processus biophysiques. Il n'y a pas de sémantique supportant la description de chaque instance des concepts de Crop2ML. Par exemple, il n'y a pas de convention partagée pour nommer les variables du modèle. L'intégration d'un composant Crop2ML dans un autre composant ou une autre plateforme nous oblige donc à adapter le nom de ses variables. Ce problème nécessiterait l'annotation sémantique des modèles Crop2ML, c'est-à-dire associer à la spécification aux spécifications des composants une ou des ontologies. Cela favorisera une composition sémantique des modèles. Notre perspective est de fournir une annotation de modèles Crop2ML basée sur une exigence minimale d'annotation de chaque concept des modèles Crop2ML avec des informations pertinentes pour éviter l'utilisation abusive des composants, et de permettre la distribution des modèles Crop2ML via un dépôt partagé, comme BioModels (Glont et al., 2018; Le Novere, 2006).

Table of contents

Remerciements ... iii
List of Abbreviations ... v
Glossary ... vi
Source of Funding ... vii
Résumé ... viii
Abstract ... viii
Résumé étendu ... ix
Chapter 1. Introduction ... 1
1.1. Research context ... 1
1.2. Research objectives and questions ... 4
1.3. Research strategy ... 5
1.4. Contributions ... 6
1.5. Outline ... 8
Chapter 2. State of the art ... 10
2.1. Process-based crop models ... 10
2.2. Software reuse ... 17
2.3. Crop model reuse ... 24
2.4. Conclusion ... 27
Chapter 3. Crop2ML: An open-source multi-language modeling framework for the exchange and reuse of crop model components ... 29
Chapter 4. Reuse of process-based models: automatic transformation into many programming languages and simulation platforms ... 66
Chapter 5. Crop modeling frameworks interoperability through bidirectional transformation ... 111
Chapter 6. General Discussion ... 130
6.1. Research findings ... 130
6.2. Future research directions ... 135
Conclusion ... 138
References ... 141

Chapter 1. Introduction

Modeling and Simulation (M&S) is a well-known research domain that supports the integration of knowledge of other disciplines needed in research and applications (Banks, 2010). It offers a methodology to guide modelers that includes four main steps to address a domain modelling: the definition of the modeling problem, its conceptualization, simulation, and experimentation. The first step consists to define clearly the purpose of the modeling with its requirements. The conceptualization provides the conceptual model. Then a simulation model is implemented from the conceptual model, and, finally, the simulation model is executed with different experiments to produce simulation results. For more than six decades, researchers in plant and crop science have increasingly used Process-Based crop Models (PBM) to primarily increase scientific knowledge underlying the dynamics of bio-physical processes involved in plant and crop growth. Currently, a plethora of PBM exists. They have been impacted by the progress in Software engineering. Different crop modeling groups have emerged and have provided different modeling and simulation frameworks. The difference between these frameworks prevent researchers from realizing the potential benefit of PBM reuse between them. For my thesis, I am interested in the reuse of the PBM components between different crop modeling groups. The outline of this chapter is as follows: The following section provides the motivation for research and the identified research issues. Section 1.2 details the research objective and questions. The research strategy is explained in Section 1.3. Section 1.4 presents the main contributions of this thesis. Finally, the outline of the thesis is given in Section 1.5.

1.1. Research context

Several factors have motivated the exchange and the reuse of PBM components between crop modeling groups.

1.1.1. Motivation for research

The increasing number of PBM is interesting but raises great challenges on their reuse in different crop modeling groups. PBM are essentially computer tools used to codify the plant and crop growth and development theory. They have been increasingly developed and continuously expanded to meet a wide range of applications. Apart from their primary role, they are used, among others, to (1) analyze the interaction between the plants and their environment (2) optimize farmers' strategies (3) assess agroclimatic risks and make technical or political decisions (4) estimate and predict agricultural yields. They allow researchers to examine scientific hypotheses (K. Boote et al., 1996) or to analyze and predict

the response of agricultural systems to climatic (Porter et al., 2014), agronomic and more recently genotypic factors. They can also support the analysis of several plant physiological traits and experiments, which could not be realized in the field.

Moreover, digital agriculture helps researchers improve knowledge of plant growth and development through the mass of data produced by the close range or remote sensors. It raises new issues that lead to question the formalism of the models, in particular, by modeling more precisely the causal mechanisms of the plant's responses to its environment. In parallel, the advance in Software engineering is also one of the significant factors of PBM increase by promoting different implementation approaches. It gives modelers the capability to define generic crop models based on common physiological functions involved in the growth and development of a wide variety of crops (e.g. AquaCrop (Steduto et al., 2009), SPASS (Wang & Engel, 2000), STICS (Brisson et al., 1998)). To overcome the model reuse issues and other problems related to model building, conceptual frameworks have been built and have highly facilitated the development of multiple models for the same crops. We can mention, among others, the frameworks RECORD (J. E. Bergez et al., 2013), BioMA (Donatelli et al., 2012), OpenAlea (Pradal *et al.*, 2008, 2015), SIMPLACE (Gaiser et al., 2013), APSIM (H. E. Brown et al., 2014), DSSAT (Hoogenboom et al., 2019), CROSPAL (Adam, 2010a). The synergy between modeling frameworks has highly been desired for improving crop models by sharing, model components and concepts and continuing testing them (Stöckle & Kemanian, 2020). Despite their interest and advances, the differences in frameworks have negatively affected the advances in crop modeling. They cause a loss of transparency for modelers, which has resulted in slowing down the development of new formalisms due to a lack of scalable standards for model development and reuse. Model components are not reusable outside the specific framework in which they have been developed, and little advantage is taken from existing model components provided by other frameworks. However, the crop modeling community has a growing need for a common approach that will help to exchange and reuse efficiently and seamlessly their model components (Holzworth *et al.*, 2014a; Martre *et al.*, 2018). This common approach is crucial for accelerating research on crop modeling, to increase the degrees of explanatory mechanisms (Antle et al., 2017), and benefit from collaborative modeling. Additionally, it must provide the potential to test seamlessly alternative hypotheses provided from different modeling groups. It must be well defined to allow for the modularity, reproducibility, verification, validation, and reusability of crop model components. A centralized framework is a means to address all of these requirements. Modularity consists of breaking the process down into manageable and reusable small functions. Reproducibility is ensured regardless of crop simulation framework. Verification ensures that the modeled process is well designed. Validation allows testing the outputs against expected values for each modeled process independently of the crop simulation framework, and Reusability is the capability to integrate a component in a framework other than the one in which it was developed. We assume that these criteria are w

1.1.2. Problem statement

The need for modularity has long been recognized by crop modelers (Donatelli et al., 2014; Timlin et al., 1996). It allows comparing or using alternative model components with different levels of detail. It is one of the factors that permits to a third party to reuse a model component ensuring that its integration in a large component is coherent. It facilitates model maintenance at implementation level (Antle et al., 2017). These interests show the crucial importance of modularity in reuse, which needs to be carried out transparently. Modularity is addressed at implementation level in different programming languages, and the assumptions are seldom included into model components (Athanasiadis & Villa, 2013). Component reuse requires the knowledge of the modeling project objectives and a level of abstraction higher than the implementation level. In M&S, the importance of conceptual modeling is well demonstrated. A conceptual model allows modelers involved in a simulation project to understand and discuss the structure of the model (processes hierarchy) without focusing on its implementation. Its development is relevant for expressing the requirements of the modeling project. Currently, a majority of crop modelers still represents crop models or components at implementation level only, and do not use conceptual model as a reusable artefact. Conceptual models often remain informal or in mind. Moreover, the publication of crop models or components in scientific journals does not provide sufficient description associated with the modeled process which are a fundamental criterion for their reuse (Keller & Dungan, 1999). This raises the problem of reproducibility and reliability of scientific results that are strongly linked to the platform in which a component has been implemented and tested (Cohen-Boulakia et al., 2017; Hinsen, 2016). There is an urgent need to represent explicitly conceptual models, and to verify and validate models at conceptual level.

Issue 1. There is a lack of explicit representation of the conceptual models in the modeling of biophysical process. In other words, there is no separation between the domain space and the implementation.

In the same way, given that modeling solutions or model components are often provided at implementation level, the reuse approaches are based on their use as black box (Rizzoli *et al.*, 2008) or their re-encoding to be conform to the target platform requirements. The black box approach makes possible the coupling of components through a wrapping system (encapsulating the component in a new architecture). This technique requires the implementation of complex algorithms. It obstructs the internal behavior of the component and does not enable the extension or renovation of the component. From a social point of view, a black box component is less likely to be trusted (Janssen et al., 2017). Re-encoding a component requires programming skills for each language in which the component is implemented. The lack of an automatic transformation system can increase development time and cost.

A transformation system is needed to ensure that the simulation models (source code) and the conceptual models are consistent.

Issue 2. Existing PBM frameworks do not provide a transformation system for model component reuse.

1.2. Research objectives and questions

We formulate our research hypothesis based on the background and our analysis of existing PBM components, and the frameworks in which they have been implemented:

The definition of shared modeling concepts to represent biophysical processes involved in the crop growth and development allow component reuse between modeling and simulation platforms.

The main objective of this research is to design a multilanguage framework for PBM components exchange and reuse. The framework will provide a novel approach shared by PBM frameworks to describe conceptual models (to address issue 1), for performing model transformations to different languages and frameworks, and supporting consistency between a conceptual model and its implementations (to address issue 2). Consistency refers to maintaining the compatibility between a conceptual model and its implementations. That is, the framework should allow generating a conceptual model from model implementation. The research aims to contribute in the exchange and reuse between crop modeling frameworks. To address the above issues on reuse of PBM component and to achieve our research objective, two main research questions are investigated.

Question 1. Is there a shared representation of PBM components between crop modeling and simulation frameworks?

- **Q.1.1.** What is the optimal level of abstraction of PBM components?
- **Q.1.2.** What are the main requirements for achieving the transformation between the specification modeling language and PBM frameworks?

Question 2. How can we design a transformation system to achieve the research objective?

- **Q.2.1.** What functionalities should the transformation system provide to generate model component that conform to PBM frameworks?
- **Q.2.2.** How to maintain consistency between a conceptual crop model component and its implementation?

Crop modeling and simulation frameworks use a simulation engine, a mechanism that links input data and the PBM to produce simulation results. Different simulation engines can be based on different

models of computation (MoC) such as dataflow, DEVS simulation (scheduling events), control flow, used to coordinate the execution of the model. The interactions between the physical and biological components of the biophysical system are also managed through the simulation engine that defines the execution logic of the components. Moreover, the biophysical system is often linked to a decisional system, modeled through different formalisms (agent-based, rules) that also affects the simulation results. The objective of this research is neither to provide a crop modeling platform nor to provide the execution logic of the set of PBM components of a modeling solution (that is, of a crop model). The aim is to define a centralized framework to describe PBM components, which can be sequentially composed and reused transparently and automatically in existing modeling and simulation frameworks.

1.3. Research strategy

In order to follow an appropriate research strategy, it is important to define the philosophy of research. Three main interrelated components composed the research philosophy: ontology, epistemology and methodology (Tolk, 2016).

Ontology relates to the study of what exists. The ontological paradigms are mainly realism, critical realism, and pragmatic realism. Realism considers that reality exists and is independent of the observer. In the critical realism, reality exists but the knowledge that one develops of reality is dependent on the observer (Guba & Lincoln, 1994). Pragmatic realism brings a moderate form of the realism assuming that reality exists but it derives from the usage made of it. Therefore, reality is function of actor's belief and truth is a function of actor's practices (McCarthy, 1996). In crop modeling, even if the real is known, the knowledge produced from crop modeling depends on the modeling hypothesis formulated by the modeler. Therefore, the critical realism is the ontology approach used in this interdisciplinary research.

Epistemology refers to how we define and represent knowledge. The common epistemology paradigms are positivism, interpretivism and postpositivism (Bunniss & Kelly, 2010). Interpretivism is opposite to positivism and their differences are well illustrated in (Huang, 2013). Positivism states that all knowledge about reality is exclusively based on experiments and observations independently from observer's perceptions (Clark, 1998). Interpretivism focuses on understanding of the observer's interpretations and values subjectivity. Reality is relative and truth is constructed from observers' perceptions. In postpositivism, the objectivity remains an ideal to achieve but the production of knowledge is influenced by actors' perceptions. So, it is possible to have different conceptualizations of truth. That is adopted in this research.

Methodology refers to how knowledge is applied to provide efficient methods which will be followed during the research. We distinguish different ways to characterize the type of research methods: inductive/deductive, qualitative/quantitative, applied/fundamental. The inductive method

starts with specification and produce generalization (e.g., all the PBM components we use are discrete-time; therefore, all PBM components are discrete-time). The deductive method starts with generalization and produces a specification (e.g., all crop modeling frameworks depend on programming languages. BioMA is a crop modeling framework; therefore, BioMA depends on programming languages). The deductive method is used to test hypotheses while the inductive method generates hypotheses. The two mixed reasoning methods are in line with our research, from inductive (define hypotheses from our first knowledge) to deductive (test hypotheses). The qualitative method is primarily exploratory and helps to develop ideas or hypotheses based on qualitative data while quantitative method includes survey methods, numerical methods, etc. This research uses mixed methods (qualitative/quantitative) as it started by identifying the qualitative differences between existing modeling approaches and use numerical methods (difference equations), design patterns to reach the research objective. Moreover, it is an applied research.

We assume that the reuse experience relies on a shared resource, which in our case is the common concepts to represent a crop biophysical process. Thus, having these shared concepts make it possible to represent a process regardless of modeling platforms to support the exchange and reuse of components. However, other factors are needed to achieve the reuse objective. Among these factors, the adoption of the modeling language by crop modelers and the reuse proof-of-concept. To achieve that, this research is conducted as part of an international consortium of large modeling and simulation platforms called AMEI (Agricultural Modeling Exchange Initiative). This consortium is a source of our knowledge on crop modeling frameworks (meetings, survey methods, tests) and allows us to apply our approach.

Based on the research philosophy adopted and research questions, the adopted research strategy is classically described as follows:

(1) identification of the issues, research objective and questions;
(2) presentation of the background to produce the research hypotheses;
(3) conceptualization of the approach and implementation;
(4) proof of concept to test the hypotheses;
(5) improving the approach by an iterative process (3) to (5);
(6) results.

Our approach primarily focuses on widely used crop modeling and simulation platforms (DSSAT, BioMA, SIMPLACE, Record, OpenAlea) and its extensibility has been proven subsequently. We used three PBM components as a case study to develop and evaluate our approach.

1.4. Contributions

The main contributions of this thesis are to:

Propose of a shared crop modeling language (Crop2ML) for model specification: We address the lack of representation of conceptual models and the diverse interpretation of modularity in PBM frameworks. For that, we propose a set of generic concepts that allow representing shared conceptual models between modeling frameworks. Based on these concepts, we design a declarative language with a modular approach to describe the specifications of model units and their composition. The concepts contain all information that ensure the provenance of the component and allow representing it regardless of the specificities of PBM frameworks. Our representation of model components increases their portability, model reasoning and follows FAIR principles for software (Lamprecht et al., 2019). Moreover, this modeling language contains unit tests concepts that will help modelers to integrate unit test into model development.

Separate model specifications from their implementation: With our approach, we keep the essential information to represent a component and hide implementation details. Model implementation derived from model specifications. This improves understanding of the models, and improves collaborations between modeling groups during modeling activities due to the fact that our model component format offers a model structure independent of implementation frameworks with the capacity to provide information searching (parameters, state variables, etc.), and to integrate them into a structured model catalogs.

Provide a transformation system between crop modeling frameworks: The main mechanism often used to address model component reuse was to re-encode model or wrap it with the requirements of the target platforms. If the first approach is cumbersome and requires high skills in different languages and framework specificities, the second obstructs the knowledge behind a component and uses it as a black-box. We address the reuse issue with a white-box approach. For that, we define a small language that 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 way to describe a process with the capacity to be integrated automatically in various platforms. A transformation system was built, and it 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 platform 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. We have also extended this transformation system with the capabilities to infer Crop2ML from a framework model component under some constraints in

order to avoid building from scratch an existing PBM component. Thus, the extended transformation system led to an interoperability system between PBM frameworks.

Design and implement a framework for model component exchange and reuse between crop modeling and simulation frameworks: We tested the applicability of the proposed approach by developing a framework. This framework bridges the gap between modeling and simulation frameworks in model component reuse and manages model lifecycle (creation, edition, composition, transformation, verification, validation).

1.5. Outline

The remainder of this document is organized as follows:

Chapter 2 presents the state of the art related to our work on PBM reuse. We begin with the description and diversity of the existing models. Then, we draw up a state of the categories of software reuse with a focus on some examples such as software components, design patterns, domain-specific languages, and transformation systems. We extend with the particularities in model reuse and some initiatives related to model exchange and reuse. This chapter has been conducted to help position this research with regard to the related work in model reuse by identifying the limits of existing modeling frameworks and the recommendations to achieve our objective.

Chapter 3 provides an outlook of the multilanguage modeling framework as a new approach for bridging the gap between crop modeling and simulation frameworks. It essentially focuses on the main concepts of the conceptual modeling language and describes the main components of this framework. It emphasizes the requirements of import and export between the conceptual language and PBM frameworks.

Chapter 4 presents the use of an embedded domain specific language with the framework. It also describes the design and implementation of a transformation system that transforms from the conceptual models to various languages and frameworks.

Chapter 5 aims to bring the consistency between the conceptual model and the simulation model (code source) to end up with an interoperable system.

Finally, Chapter 6 presents the evaluation of the research study and the future works.

Publications in peer reviewed journals

Midingoyi CA, Pradal C, Athanasiadis IN, Donatelli M, Enders A, Fumagalli D, Garcia F, Holzworth D, Hoogenboom G, Porter C, Raynal H, Thorburn P, Martre P (2020) Reuse of process-based models: Automatic transformation into many programming languages and simulation platforms. *in silico Plants,* diaa007. https://doi.org/10.1093/insilicoplants/diaa007

Midingoyi CA, Pradal C, Athanasiadis IN, Donatelli M, Enders A, Fumagalli D, Garcia F, Holzworth D, Hoogenboom G, Porter C, Raynal H, Thorburn P, Martre P (2021) Crop2ML: An open-source multi-language modeling framework for the exchange and reuse of crop model components. *Environmental Modelling and Software*. (*in Press*)

Publications in preparation

Midingoyi CA, Pradal C, Athanasiadis IN, Donatelli M, Enders A, Fumagalli D, Garcia F, Holzworth D, Hoogenboom G, Porter C, Raynal H, Thorburn P, Martre P (na) Crop modeling frameworks interoperability through bidirectional transformation. In preparation for publication in *Environmental Modelling and Software*.

Communications in international congresses and symposium

Midingoyi CA, Pradal C, Enders A, Fumagalli D, Raynal H, Athanasiadis IN, Porter C, Hoogenboom G, Holzworth D, Garcia F, Thorburn P, Donatelli M, Martre P (2019) Crop2ML: a crop modeling metalanguage shared between different crop simulation platforms. *In* Annual main meeting of the Society for Experimental Biology (SEB2019), 2-5 July 2019, Seville, Spain. ⟨hal-02981423⟩ (oral presentation)

Martre P, Midingoyi CA, Enders A, Fumagalli D, Raynal H, Athanasiadis IN, Porter C, Hoogenboom G, Holzworth D, Garcia F, Thorburn P, Donatelli M, Pradal C (2020) Crop2ML: A Crop Modelling MetaLanguage shared between different crop simulation platforms. *In* International Crop Modelling Symposium, Crop Modelling for the Future (iCROPM2020), 3-5 February 2020, Montpellier, France. (Keynote presentation)

Midingoyi CA, Pradal C, Martre P (2020) Enhancing biophysical processes reuse across multiple crop simulation platforms via CyML language and transpiler. *In* 10th International Congress on Environmental Modelling and Software (iEMS2020), 14-18 September 2020, Brussels, Belgium. (oral presentation)

Midingoyi CA, Martre P, Pradal P (2020) CyML language and transpiler: Re-using biophysical processes in crop growth models across multiple platforms. *In* International Crop Modelling Symposium, Crop Modelling for the Future (iCROPM2020), 3-5 February 2020, Montpellier, France. (poster presentation)

Chapter 2. State of the art

To address the objective of the research framed in Chapter 1, we need to define a system for biophysical process component exchange and reuse between crop modeling platforms. This system aims to allow a model builder to seamlessly integrate a component in its platform, improve it if needed, and combine it with other ones, and export it for further reuse.

The reuse of a biophysical process component requires knowing the features of process-based crop models (PBM). The advances in software engineering offers the potential for various implementations of PBM. First, this chapter describes PBMs and their evolution with a focus on the reuse needs. Software reuse refers to the process of creating or extending software systems from existing ones in a “cost-efficient way” (Kim, 2009) rather than building them from scratch (Krueger, 1992). Software reuse is a popular approach to lower the cost and time of software development while ensuring better software quality and reliability (Rico et al., 2008). Although software reuse has been addressed since the birth of software engineering field in 1968 (Krueger, 1992), this research domain is still evolving. The needs for reuse have been expanded from software requirements to source code. Thus, software reuse includes the entire range of M&S activities, such as requirements, specifications, tests, documentation related to the modeled system, and the code describing its dynamic. It is interesting to follow M&S principles to tackle model reuse in order to make reusable the result of each step of the modeling process. Abstraction is a key element in model reuse (Athanasiadis & Villa, 2013; Robinson et al., 2004). Therefore, in the section 2.1, we analyzed the domain, crop growth biophysical processes, to define the good abstraction to represent a process regardless of the PBM platform. Combining M&S principles and Software reuse will help to formalize crop model development for process reuse. This research is based on terms, which may have various meanings in other literatures. It is therefore necessary throughout the manuscript to clarify our definitions.

2.1. Process-based crop models

2.1.1. Crop growth and development

A crop is defined as a collection of individual plants of one or more species grown in a unit area. Crop growth refers to an irreversible increase in size (mass, length, area, volume etc.) whereas its development represents the continuous change in plant form due to the appearance of new organs. Crop development is a discrete phenomenon often characterized in terms of the duration of appearance of new organs. Different biophysical processes are involved in crop growth and development. They are mainly influenced by the crop environment and soil processes. Thus, to improve the knowledge of these

processes a system approach (Von Bertalanffy 1969) is used to define a set of interrelated parts (components) in the soil-plant-atmosphere continuum (system) that operate together for a common purpose. Each of these parts can represent another system (Klir, 2001), and the soil-plant-atmosphere system can be represented hierarchically until a primitive part is reached. Change in one system's component produces changes in other components because of the interactions. The soil-plant-atmosphere continuum can be viewed as a system due to the interactions among the soil, the atmosphere, and the plants that live in it. The behavior of this continuum may also be changed by crop management practices. The complexity of these interactions does not facilitate a direct understanding of crop functioning, and requires the use of integrative simulation models.

2.1.2. What is a model?

Model is a polysemous term and should be defined in the context of our work. We have retained three definitions that fit with our problem:

- a model is an "abstraction of a system intended to replicate some properties of that system" (Page, 1994)
- a model is defined as "a mathematical representation of a system" (Jones & Luyten, 1998)
- a model is a "description of (a part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer" (Kleppe et al., 2003).

The first definition tackles the notion of abstraction. It refers to simplifications that offer a comprehensive description of the system based on specific objectives. It reveals that, at the same time, different models can be derived from the same system. Due to the complexity of the system behavior influenced by crop management practices and environmental conditions, it is not a straightforward task to produce a comprehensible and operational representation of a crop (Murthy, 2004). We can retain that a crop model is based on a system, a selection of properties of the system, and that it can replace the system for a specific objective. The second definition adds the notion of representation and emphasizes that the model can be based on mathematical theory (differential equation, graph theory). The last definition treats model as a comprehensible object by a computer. It can be therefore a source code. These three definitions can be interpreted as different modeling levels where for each level a model can be defined (from requirements to source code).

2.1.3. Crop modeling evolution

Crop modeling science has extensively been developed over the past 60 years in parallel with our knowledge in crop physiology and computer science. Existing crop models are based on a diversity of modeling approaches with different levels of details (Hammer et al., 2019). The pioneering work of de Wit (1965) attempted to model crop photosynthesis. The results obtained from this model led to a high diversity of advanced models (see Chapter 2 Fig. 1). Some of them more oriented toward the crop scale, such as CERES (Ritchie & Otter, 1985) CropSyst (Stöckle et al., 2003), SUCROS (Van Ittersum et al., 2003) or APSIM (Keating *et al.*, 2003; Holzworth *et al.*, 2014b) and others more oriented toward landscape and regional scales such as EPIC (Sharpley & Williams, 1990), LPjML (Von Bloh et al., 2018), or GLAM (Challinor et al., 2004). The existing models are suitable for a wide range of applications under different environmental conditions where processes are co-regulated by environmental factors such as water and nitrogen (Muller & Martre, 2019). The high diversity PBM (structure, complexity) originates from the modeling purpose that allows their classification in different categories. (Jones et al., 2017; Muller & Martre, 2019)

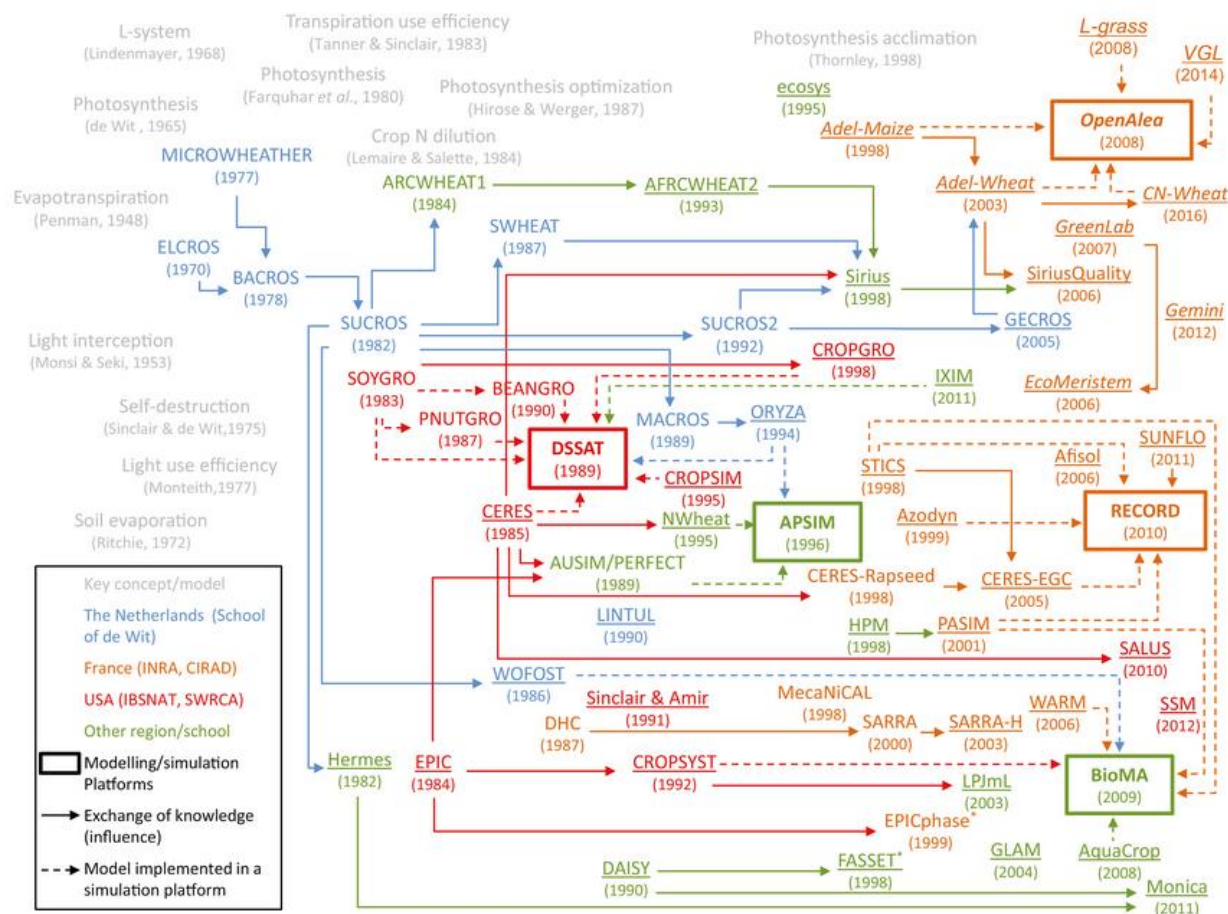

Figure 1: Illustration of the historical evolution of the main process-based crop models. Reproduced from Muller and Martre (2019).

2.1.4. Classification of models

Models can be classified in many ways. The models that deal with crop growth and development can be distinguished into different categories according to the modeling purposes (analysis, description, prediction, exploration) (Singh, 1994; Van Ittersum et al., 2003) . Here are few of them:

a. **Statistical / Mechanistic models**: Statistical models express the relationships (step down regressions, correlation, etc.) between model variables (Lobell & Asseng, 2017), whereas mechanistic models are based on the knowledge of the underlying processes of the system and try to explain the influence of the driven variables on the outputs of the model.

b. **Deterministic / Stochastic models**: Deterministic models ignore random variation, and predict the same outcome from a given starting point whereas a probability is attached to model output in the case of stochastic models. The latter may predict output distribution.

c. **Descriptive / explanatory model**: A descriptive or empirical model defines the behavior of a system in a simple way (Singh, 1994). The model reflects little or none of the mechanisms that are the causes of phenomena. An explanatory model provides a prediction and an explanation of integrated behavior from more detailed knowledge of the underlying physiological processes. Processes can be quantified separately and interconnected to analyze emerging properties of the system that can be deduced from the individual processes.

d. **Static / Dynamic models**: static models involve no concept of time or describe a system at a given time point, unlike dynamic model where states variables change with respect to time.

e. **Discrete / Continuous model**: In a discrete model, state variables take values at particular points in time whereas in a continuous model, state variables change continuously with respect to time.

Each type of model has its interest according to the modeling purpose. Jones et al. (2017) present two examples showing the interest to choose a descriptive or a statistical model. A descriptive model is required to describe how agricultural systems respond to the external environmental drivers as well as decisions or policies under consideration. A statistical model can be used to predict crop yield at regional scale based on observed climate variables and crop regional yield statistics over multiple years. However, models that increase the scientific knowledge involved in plant and crop growth are the basis of decision modeling approach and we need to focus on them to increase the underlying science. They are called process-based models (PBM). Crop growth occurs over time within the growing season, consequently PBM are inherently dynamic. Due to the fact that PBM are often driven by discrete variables, the system is numerically analyzed with discrete time. A general form of dynamic system in discrete time described in Wallach *et al.*, (2006) is:

$$U_i\,(t + \Delta t) = U_i\,(t) + f_i\,[U(t), X(t), \theta] \qquad \text{with } i = 1, \dots, n$$

where,

- t is time;
- Δt is the time step;
- $U(t)$ is the vector of state variables at time t;
- $X(t)$ is the vector of explanatory variables at time t;
- θ is the vector of parameters;

The time constant is not the same for all modelled processes in PBM. Even though PBM can be, in some instances, represented by finite difference equations formalism, it is common practice to use other state variables (sometimes categorical variables) or control structures to describe procedurally the PBM algorithms.

Most of them have been developed in the frame of one-dimensional crop-soil-atmosphere system with an emphasis on vertical fluxes of energy, water, C, N and nutrients between the atmosphere, plant and soil root zone continuum (Jones et al. 2017). In this thesis, we also focus on mechanistic and deterministic models in the aim to increase the science beyond crop growth even if most crop models represent a compromise between these mechanistic and empirical modeling (Yin & Struik, 2015).

2.1.5. Model complexity

The level of complexity of PBM depends on the objective of the modeling exercise and the degree of biological details and realism they represent. An effective way to tackle this complexity is to design PBM based on a top-down approach (Hammer 1998). This approach consists in representing the overall vision of the studied system. Then the general processes are distinguished, eventually broken down into sub-processes, and so on until the simplest processes is obtained. Once the simplest processes are established, they are grouped together to form a whole. This approach requires prior knowledge of lower-level parts. An important point raised by modelers is that a model should be built, in such a way that its components links can be removed and changed by better relationships among processes (Dourado-Neto et al., 1998; Giller et al., 2011; Monteith, 1996). The majority of existing PBM share the common approach of decomposition based on the functional processes that govern crop growth and development, which means that there are roughly composed of potential interchangeable parts (Adam *et al.*, 2010a). However, these key physiological processes (see Chapter 2 Fig. 2), e.g. phenology, biomass accumulation, yield formation, and water and nutrient uptake differ in their modeling approach (Asseng et al., 2013).

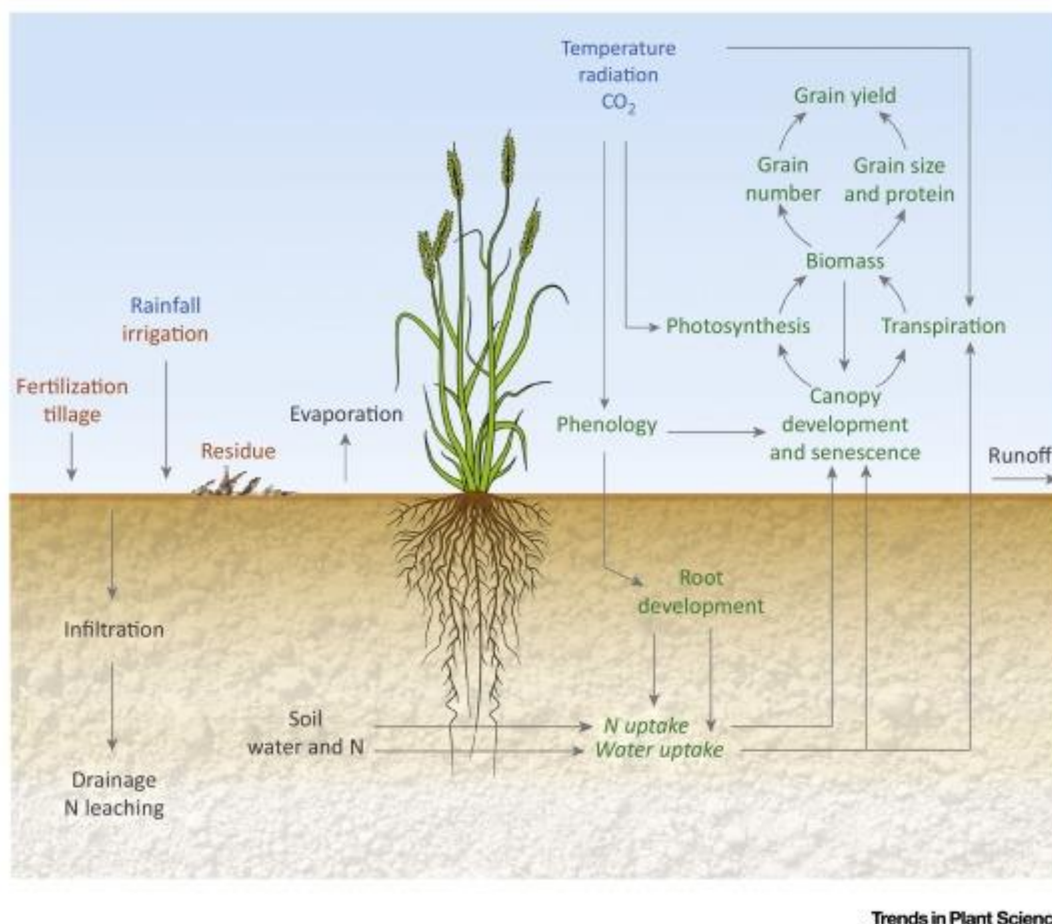

Figure 2: A simplified schema of crop model showing the key processes involved in crop growth and development and their interactions with the crop system. Reproduced from Chenu et al. (2017).

2.1.6. Crop model intercomparison and improvement

Two levels of improvement are required:

Enhance the science underlying crop growth and development

Model improvement can be achieved by improving process algorithms and the interactions of these individual processes (Yin & van Laar, 2005) to align with the requirements of the development of next-generation crop models (Rosenzweig et al., 2013). These requirements include the need to have better responses to stress, climate variability and climate change (Antle et al., 2017). PBM are continually being improved to bridge the gaps between genotypes and phenotypes (Masseroli et al., 2016; Muller & Martre, 2019; Wang et al., 2019; Yin & Struik, 2015) and with management practices (Stöckle & Kemanian, 2020). Several research projects promote international collaboration focusing on the comparison of model outputs from different modeling groups and against global experimental datasets. For instance, Palosuo et al. (2011) compared the performance of eight widely used crop growth simulation models (APES, CROPSYST, DAISY, DSSAT, FASSET, HERMES, STICS and WOFOST)

for winter wheat. This study revealed that none of the models perfectly reproduced recorded yields at all the sites in all the years. It stated that a good crop yield prediction for some models was at the expense of overestimating or underestimating the harvest index or total biomass. An intercomparison study of 29 wheat models as part of Agricultural Model Intercomparison Improvement (AgMIP) project (Rosenzweig et al., 2013) also showed uncertainty in yield simulation mainly due to the temperature response functions in the models (Asseng et al., 2013; Rötter et al., 2011; Wang et al., 2017). However, it was recognized that these intercomparison activities did not really advance the understanding of different crop models across the agricultural modeling community. The improvement of both individual crop models and ensembles of multiple models for a particular crop (Asseng et al., 2013) remains a crucial challenge.

Improve model development to align with FAIR principles

To achieve the goal of models intercomparison research projects, in terms of model improvement, the specification and implementation of PBM need to follow FAIR principles in order to facilitate the integration of alternatives components provided by different modeling groups. The different recommendations for the next generation of PBM (Antle et al., 2017) to facilitate model integration and reuse can be summarized as:

a. the different processes must be identified as well as their interactions;

b. a need for investments in the design of modular model component;

c. mathematical and logical formalisms must be identified to have a good knowledge of the modeling assumptions.

Currently, PBM are referenced as computer codes at a low-level of abstraction, that hides the formalism behind the biophysical processes. There is currently little exchange and reuse of PBM components between different modeling groups despite theoretical and application interests (Holzworth et al. 2014). The main limitation comes from compatibility issues between different modeling approaches and implementations. Model reusability remains a challenging task and it often forces modelers to rewrite manually the code from other models. This representation of PBM do not allow aligning FAIR principle in terms of having findable, accessible, interoperable and reusable components.

Data harmonization is the current approach used in AgMIP to address multiple models use (Porter et al., 2014). A data exchange mechanism with model variables is defined in accordance with international standards through data interoperability tools. It supports a flexible data schema with methods to fill gaps in data (Janssen et al., 2017). Researchers and modelers are able to use these tools to run an ensemble of models on a single, harmonized dataset. This allows them to compare models directly, leading ultimately to model improvements. Data harmonization is crucial for model simulation

with data provided by other groups or for multi-model ensembles or intercomparison activities. However, even if uncertainties are observed, this approach does not allow finding the process, which essentially impacts the simulation results. It does not therefore allow improving individual crop models.

2.2. Software reuse

Several approaches of software reuse exist (Krueger, 1992). Their differences are mainly related to the way they abstract, select, maintain and integrate (Biggerstaff, 1989) software artifacts in the reuse process. Abstraction is the main factor. Software artefact concerns any entity (source code, documentation, specifications, transformation, etc.) which can be used, modified or created during software development. All technologies to implement a software reuse approach provide an integration framework required to *c*ombine a collection of selected and specialized artifacts into a complete software system with the appropriate exports and imports (Krueger, 1992).

In his survey of reuse, Krueger (1992) identified several categories of reuse approaches along the four aforementioned dimensions introduced in Biggerstaff (1989). This section reviews some of these categories and extends them with recent literature.

2.2.1. Software reuse approaches

High-level and very-high-level languages

A language is based on a grammar which defines its syntax. The syntax defines the form of valid expressions used in the language, while its semantic defines the meaning of the expressions. We distinguish human languages, i.e. the ones used for human communication, from computer languages comprehensible by the machine. Here, we focus on the computer languages.

Formally, a grammar consists of four parts (Slonneger & Kurtz, 1995):

$G = < \mathrm{T}, N, P, S >$

where,

- T is a finite set of terminal symbols representing the alphabet of the language that can be combined to form the expressions (terms, sentences) of the language;
- N is a finite set of non-terminal symbols called syntactic categories that are some parts of the expressions;
- P is a set of production rules which are pairs formed by a non-terminal symbol with a series of terminal symbols and non-terminal symbols. The choice of non-terminals defines the expressions to which a meaning is ascribed;

- *S* is an element which represents the start symbol (e.g., program) that imposes the order of production rules.

There are different types of language syntaxes such as concrete and abstract syntaxes. A concrete syntax determines how to recognize the sentences in a program (Mosses, 2006). For that, a recognizer (software utility) is built to parse the program. It is usually composed of a scanner or lexical analyzer which reads and decomposes the program into a list of tokens, and a parser which produces a derivation tree, parse tree or concrete syntax tree (CST) from the generated list of tokens (see Chapter 2 Fig. 3).

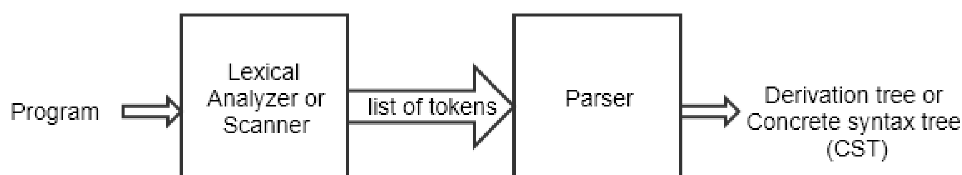

Figure 3: Process for language recognition.

The CST contains all the information in the program. However, some information is not necessary to capture its basic structure. This is the role of an abstract syntax tree (AST). The CST is therefore transformed into an AST in a much simpler form through the construction of other nodes while maintaining the structure of the program. This transformation process is also called Parser. The form of the AST can be chosen freely, and allowing proceeding to the checking of the semantics.

In **high-level languages** (HLLs), the language constructs (e.g. arithmetic expressions, iteration, and loop) are higher level than those of the earlier languages (assembly languages). However, HLLs can also produce low-level constructs depending on whether they are compiled or interpreted. Interpreted languages allow reusing code directly in any machine but they are commonly recognized to be slower. In contrary, compiled languages are generally faster but require recompilation. According to their implementation, some HLLs allow explicit datatype declaration and others support implicit datatypes. Some allow using different programming paradigms (object-oriented, procedural, functional) in software development. HLLs, such as C, C++, Java, or Python are treated as software reuse examples because they meet software reuse needs, namely speed and efficiency in software development. The main limitation of HLL comes from the fact that their abstraction is low level. That is, it is not straightforward to capture the requirements of a system through its implementation.

Very high-level languages (VHLLs) have a higher level of abstraction than HLLs. They define specifications, definitions or descriptions. They can be more expressive than HLLs since they follow the domain abstractions and semantics as closely as possible. Domain-Specific Languages (DSLs) are a type VHLL of particular interest for our work. DSLs have several names related to their representation.

We can cite, among others, Little Languages, Micro Languages, Domain Modeling Languages (DMLs), Domain-Specific Modeling Languages (DSMLs), Domain-Specific Visual Languages (DSVLs; de Lara and Vangheluwe 2004), Domain-Specific Visual Modeling Languages (DSVMLs), or Domain-Specific Embedded Languages (DSELs) according to Mernik et al. (2005). DSLs allow solving a category of problems in a specific domain. They concern many domains (biology, computer science…). DSLs are not new and exist since the early programming languages; for example, Fortran specifically has been built for numerical computation and evolved into a general purpose language. HTML is a language specified for web page creation. Significant research efforts have been focused on DSL development (Degueule et al., 2015; Kurtev et al., 2006). Mernik et al. (2005) identified a set of patterns in the workflow of DSL development (decision, analysis, design, and implementation phases) extending thus the earlier work of Spinellis (2001).

Different methods of DSL design have been explored, such as language specialization and language extension. These two methods allow avoiding building a DSL from scratch. Language specialization pattern consists in removing unnecessary features of an existing language (base language) to meet the given requirements and defines a DSEL (Hudak 1996). A language processor can check DSEL conformance to guarantee the removal. The language extension pattern consists in adding new features to an existing language. The new DSL inherits the syntax and semantics of the existing languages and can include other semantic or features such as new built-in functions or datatypes. XML-based DSL is another approach of DSL design where the grammar is described by a Document Type Definition (DTD) or XML schema (Parigot & Courbis, 2006).

The main challenge for DSLs is to maintain their specificity to respond to the evolution of the domain. This requires defining the right level of abstraction that specifies the concepts in the domain, and that can be easily adapted to keep up with the changes in the domain. This is of course more difficult to do when it is a multidisciplinary domain.

Software component

A software component (SC) is basically a software unit with a well-defined interface and explicitly specified dependencies. A SC can be as small as a block of reusable code, or it can be as big as an entire application. It can be easily plugged together with other components to form or extent a software application. SCs or building blocks are autonomous (stand-alone) units. Their reusability is based on the principles of abstraction and encapsulation with a particular structure, and are designed for a specific purpose. They help to reduce the amount of time required to design and implement a new software system. The nature of a SC is diverse and depends on the scale and the language in which it is implemented. As different levels of composability, we can cite subroutine, abstract data types, modules, packages, subsystems, classes. The best abstraction for SC reusability is its interface which embeds the

domain-specific concepts. SC interface must describe its functionality and facilitate component selection, validation and specialization. The main drawback is that interfaces may not be sufficiently detailed to satisfy reuse requirements. Moreover, if the SC abstraction is not well defined, the description of the component can often be as difficult to understand as the source code, increasing thus the intellectual effort to capture the software requirements. The level of granularity of the component is also a factor for reuse effectiveness.

Two types of components composability are distinguished and each raise different questions (Dhami 2012). A syntactic composability that asks the question whether the components can be connected and a semantic composability that asks the question whether the components that represent the composite model can be meaningfully composed. The latter depends on expert domain knowledge.

Software schema

Détienne (1991) defined software schema as a “knowledge structure which represents in a more or less abstract way, programming objects, functions and global or local strategies used in algorithms”. Unlike software components, which focus on source code reuse, the reusable schema artifacts are abstract algorithms, data structures, and higher-level abstractions. A specification of a reusable schema must contain a formal semantics description of the schema, and the conditions of its validity, whereas a realization may be a source code. Examples of this category of software reuse are design patterns (Gamma et al., 1995), UML (Unified Modeling Language) diagrams, data-intensive workflows, generative programming (e.g. templates; Stepanov and Lee 1995).

The American architect Christopher Alexander talked about patterns in buildings and towns, showing how an entity can arise from the relationships between a recurrent problem and its solution:

> “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (Alexander et al., 1977)

By analogy, in the field of reusable object-oriented software design, Gamma *et al.,* (1995) defined “design patterns” as the descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. They are therefore based on solutions implemented in object-oriented programming languages. They are characterized by their name, the problem, the solution and the consequences in terms of impact on the flexibility, extensibility, or portability of the system. Spinellis (2001) proposed three categories of design patterns for DSL: “structural”, which involves the creation of a DSL, “creational” if it describes the structure of a system involving a DSL, and “behavioural” if it describes DSL interactions. Thus, creational pattern includes language specialization and extension patterns presented above.

Data-intensive workflows or scientific workflow aims to capture a series of scientific methods, which describe the design process of computational experiments. Workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities (Rusinkiewicz & Sheth, 1995). There is a growing interest in scientific workflows to manage and share scientific computations and methods to analyze data. Even if they are supported by an execution logic (dataflow), their visual representation as a graph can be useful to describe software components assembly. The main drawback of software schema occurs when its formal specification is large and complex, making their understanding and use difficult.

Application generators

Application generators are comparable to compilers. They automatically translate input specifications into an executable program. In application generators, input specifications are very high-level of abstractions, which allows separating the system specifications from its implementation. Thus, the developer focuses on the concepts of the domain rather than how to implement the system. Application generators reduce duplications in software systems development by generalizing and embedding the commonalities of software systems. The abstraction specifications of application generators come from the application domain, and they can be presented by templates, UML diagrams or DSL. Examples of this category of software reuse are parser generators. Parser generators take as input a specification language and generate a part of or a whole source of a compiler. An example is ANTLR (Parr, 2013), a widely used parser generator in academy or industry, used to build all sorts of languages, parsers and frameworks. It reduces the complexity to generate multiple parsers from a series of grammars into the same framework.

The main advantage of application generators is their capability to map automatically the concepts of the application domain into executable software systems. However, if a particular application generator exists, it is not obvious to design a software system that responds perfectly to the expected needs. If it does not exist, it is not straightforward to build an application generator with appropriate functionality and performance.

Transformation systems

Transformation systems presented in Mens and Van Gorp (2006) are broader in scope than those described in Krueger (1992). Indeed, vertical transformations, a subset of model transformation, are mostly refinement transformations that map models based on a more abstract DSL to models based on a more concrete one, or to code based on a general-purpose programming language. With this transformation approach, software is developed in two phases: the semantic behavior of the system is described using a high level of specification language, then transformations are automatically applied

to the high-level specification to transform it into an intermediate representation with a lower level of abstraction in order to optimize its execution without changing its semantic behavior (Pradal, 2019). Mens and Van Gorp (2006) proposed a taxonomy of model transformation with a notion of model that encompasses all levels of abstraction, including the source code as a model at a low level. Transformation approaches depend on the level of abstraction of the model (source code or specification) in the source or target. Source-to-source transformation is a well-established solution used to address software reuse (Fernique & Pradal, 2018; Plaisted, 2013). It consists in transforming a source code from a high-level language to another. Currently, to the best of our knowledge, there is no solution targeting PBM component reuse using an automated source-to-source transformation system. However, different source-to-source transformation systems, both commercial (e.g. Baxter, Pidgeon, and Mehlich 2004) and open source (Quinlan & Liao, 2011), are available for different purposes. Some lessons can be learned and taken into account from these approaches even though they suffer from some limitations related to the context. Terekhov and Verhoef (2000) shed light on the realities of language transformations and warned on large-scale language transformation projects, which have often been a failure and have led to business bankruptcies. Thus, a subset of language features has often been defined to approach source-to-source transformers. Many transformers 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, 2003; Schaub & Malloy, 2016) and many-to-many (Baxter et al., 2004) transformers have been proposed. They are based on a common intermediate representation for the languages provided from their similarities. However, these approaches focus on the same programming paradigm in their transformation systems. For example, transforming from a procedural to an object-oriented program or a system of languages supporting different programming paradigms. Besides, to avoid losing assumptions or domain knowledge, in a particular domain context, it is useful to integrate domain knowledge in source-to-source transformer to generate a well-documented source code, embedding domain knowledge.

Software architecture

Li et al. (1992) defined software architecture as the organizational structure of a software system or a component. Software architectures describe how primitive entities such as functions, subroutines or objects compose a component or software system. Reuse software architectures capture the global structure of a component or software system and reapply this structure into the construction of similar ones in application domain. Examples of software architectures include database subsystems, software frameworks (e.g. a framework for a compiler or transpiler), and blackboard architecture. A framework for a transpiler provides the capacity to integrate different lexical, syntax and semantic analyses, transformation rules and source code generation. Software architectures are comparable to software

schema not for reuse of abstract algorithms or data structures but to focus on the structure of the subsystems beyond them. An example in this case is design pattern described above. Software architecture abstractions come from domain concepts that allow software developer to use them in order to instantiate and compose software architecture. The main challenge in software architecture is to make the representation explicit to support reuse.

2.2.2. Recent advances in software reuse

As can be seen, there is no clear distinction between the different categories of reuse approaches defined by Krueger (1992). That is, the abstraction specifications of one category can be defined by an example of another category. In addition, a category at a fine or large-scale may also be equivalent to another category. Many categories can also be addressed to end up with a software product or a diversity of software products in a software development approach. Some prominent examples in systems engineering are Model-Driven Engineering (MDE), Software Product Lines (SPL). MDE offers an efficient approach to represent domain concepts through models (primary artifacts) in the software development process. It combines domain-specific languages (DSL) that formalize the requirements, behavior and the structure of the system, and transformation systems used to automatically generate artifacts at different levels of abstraction (Schmidt, 2006). The Object Management Group (OMG) proposed Model-Driven Architecture (MDA) that implements Model-Driven Development principles. MDA is based on a set of standards, including the Unified Modeling Language (UML), the Meta-Object Facility (MOF), the Common Warehouse Metamodel (CWM). These standards enable the definition of modeling languages used to specify a system's structure and behavior. In order to focus on the modeling of complex systems, OMG opened up the MDA approach by providing the standardization of a profile called SysML (OMG, 2007). SysML extends the object paradigm of UML with communications oriented dataflow in the structural representations of the system. It also provides diagrams used to express parametric requirements and constraints in order to analyze complex system performance (Hardebolle, 2008). SPL is a critical development approach used to develop a diversity of high quality software products (a product family) based on their commonality and variability at a low cost and in a short timeframe. Software reuse is addressed for the generation of all the assets involved in the process of generation of a product family (Pohl et al., 2005).

2.3. Crop model reuse

2.3.1. Crop modeling and simulation frameworks

To avoid building models from scratch and to lean on good practices in software engineering, PBM modelers have developed crop modeling and simulation platforms (Argent et al., 2006). They help crop modelers to manage the system complexity and provide the possibility to reuse modules (sub-models) in different models and provide support for commonly needed services such as model calibration, sensitivity analysis and model visualization (Van Evert et al., 2005). PBM platforms are a set of software libraries, classes, and components, which can be assembled by a software developer to deliver a range of applications that use mathematical models to perform complex analysis and prognosis tasks (Rizzoli et al., 2008)

The PBM platforms facilitate model development offering a rich library of models and components. Most of them depend on the language in which they have been implemented, fix a particular structure of the model. Most of them are based on a component-based approach. They implement a specific software design and require a particular code convention. The main sections that can be found in model structure are initialization, rate calculations, and integration. Models are modularized along scientific discipline lines.

Jones *et al.* (2001) emphasized that despite the common modularity design of PBM platforms, there are many differences in how modularity was interpreted and implemented. These differences prevent modules from one group being reused by other groups without sometimes-considerable amounts of additional programming. The limitations to these modeling platforms as part of model exchange and reuse are their diversity of programming languages, the platform specificities and requirements that model developers have to understand, such as the structure of a model in each platform and the libraries of core modules or components. Model components can be reused in other models coded with the same platform. However, the reuse between platforms remains a great issue. The recurrent solution has been to use components as black box (Donatelli and Rizzoli, 2008) or to manually adapt components to the new platform. The first approach, black box mechanism, allows the implementation of complex algorithms in any language. However, it decreases model transparency hiding the understanding of the internal model behavior with a high framework invasiveness (i.e. "the amount of change required in model code to accommodate a framework" (Lloyd et al., 2011)). PBM platforms do not have a transformation system that generates components compatible with other modeling frameworks. Multilanguage and integrated modeling frameworks offer language bindings approach to allow third-party developers with a choice of languages (Lang, 2019; Villa, 2001). This approach reduces the implementation of algorithm in several languages. It overcomes the difficulty of implementing some

algorithms efficiently in high-level languages. However, the integration frameworks do not provide solution for model exchange and reuse between PBM platforms. Besides, they require the compilers for the supported languages of the frameworks, that makes complex their extensibility.

2.3.2. Model specification through declarative modeling

Most efforts to enrich model components with interface descriptions in source code or using ontology-based tools (Athanasiadis et al., 2011; Holzworth et al., 2010) have been found. However, it is useful to specify components interfaces with a high level of abstraction (A. W. Brown, 2000). Donatelli and Rizzoli (2008) suggested that modeling components should be developed with a generic interface (i.e., not framework specific) to enhance reuse opportunities and make unit testing easier to accomplish. The concepts of these specifications can be shared by the modeling platforms to allow supplying semantic behavior, making components self-described and reusable by any platform. According to (Muetzelfeldt, 2004), the declarative specifications of a component can be used as documentation, improve component knowledge, facilitate component reasoning, and present the hierarchical structure of a component. The concept of declarative programming is inspired by mathematics where it is common to state or declare what must be accomplished in terms of the problem domain, rather than giving a detailed stepwise algorithm on how to achieve the desired goal as is required when using procedural languages. It differs from black-box approach by processing information about the component to the framework. Code can be generated that will run in specific PBM platforms.

Athanasiadis and Villa (2013) introduced a roadmap to domain-specific programming languages for environmental modelling showing its advantage on cross-compilation for different environmental modelling platforms. Visual domain-specific language such as Simile (Muetzelfeldt & Massheder, 2003), Stella (B. Richmond, 2001) describe dynamic systems by set of differential equations making them readable. They provide a specific equation language to enable users to express their own functions. L-systems (Lindenmayer, 1968) proved well suited to describe models of plant development and were adapted to other languages such as the Python language (L-Py: Boudon et *al.,* 2012).

Model Driven Architecture (MDA), one framework of MDE, supports also declarative modeling to define conceptual models often using the Unified Modeling Language (UML). Papajorgji, Clark, and Jallas (2009) evaluated the application of MDA approach in crop modeling by using a type of action language to implement processes algorithms that cannot be expressed by single mathematical expression. Likewise, Barbier, Cucchi, and Hill (2015) defined a DSL for crop modeling and presented their need to propose in a future work a textual syntax to specify models' algorithms.

The declarative modeling approach is 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 (Hucka *et al.*, 2003; Cuellar *et al.*, 2006; Gleeson *et al.*, 2010). 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.

2.3.3. Model selection

Software reuse approach is not the sole necessary condition to support proper model or component reuse and integration in an existing model. Modelers need to ensure that an alternative modeling approach is compatible to the other components to which it will be linked. It raises the issue of semantic composability defined in section 3.3. However, this issue is not new and few studies have addressed it due to the difficulty to generalize an approach to represent expert domain knowledge. Ramírez et al., (2004) compared a set of models to select the model whose outputs are closest to real data. CROSPAL (Adam *et al.*, 2010b) uses agronomic expert knowledge to assist module selection for crop growth simulation based on the modeling objective and a comprehensive system analysis. Kuijpers *et al.*, (2019) proposed a common structure based on biological functionalities, which allows for a combination of components, yielding new models for tomato crop models. It defines a mathematical representation of the common structure and each realization of this representation is based on a specific objective. This approach increases the design space of models but it does not provide a common structure to design any kind of biological processes. Adam *et al.*, (2012b) proposed a protocol to support model selection that requires a better model documentation and knowledge about the model structure. According to Van Delden *et al.,* (2011) model selection depends on the availability of data and existing models that fit the purpose or can be adapted to fit the purpose, the choice of scale, the resolution and level of complexity required. All this information could also be found in model documentation. The main issue is the relationship between the documentation of the model and the model itself. The model selection goes through the conceptual model that facilitates the model assembly (Lamanda et al., 2012)and helps to define explicitly the structure of the model, improve the clarity of scientific understanding of the model (Janssen et al., 2017) and guide the choices to compose a model for a specific application (Adam *et al.*, 2012a).

Thus, an explicit representation of conceptual model with sound description of model constructs could help to select the appropriate model. However, it is useful to have a consistency between the conceptual model and the source code of the models, that requires a transformation system.

2.4. Conclusion

In this chapter, we present the state of the art in software and model reuse to address our issue concerning the exchange and reuse of Process-based model components between crop modeling platforms. It is crucial to make the distinction between software and model to focus on the modeling objectives or requirements, the model algorithms and the capacity to select the model based on its requirements. First, we have introduced the description of PBM:

- they are discrete-time, dynamic, mechanistic, explanatory;
- they are commonly decomposed along scientific discipline lines;
- processes are based on different modeling approaches with different time scales;
- the finite difference equations formalism is not sufficient to describe model algorithms, which lead to represent them by a set of statements with control structures;
- they are implemented in various crop modeling and simulation frameworks that reduce their reuse between different modeling groups;

Then we address different software reuse solutions and present the two prominent software development approaches MDE and SPL. Different categories of software reuse can be combined together to design an efficient approach of reuse. That is:

- the use of DSL to define the specifications at a high level of abstraction;
- the use of a component-based approach to build autonomous component that can be shared and described with the DSL;
- a transformation system that can embed software design of target crop modeling platforms to generate platform compatible model components.

This approach is close to the MDE approach since it is based on automatic model transformation. However, the difference can appear depending on whether the abstract syntax of the DSL is represented by a metamodel or a grammar. SPL are more concerned with productivity and is not really based on a platform-independent approach for describing the models.

Finally, we describe five widely used platforms used for PBM modeling and simulation. The differences between these platforms are related to their programming languages, software design and architectural constraints that affect model component reuse. There is a lack of clear model documentation or component interface description able to help select an appropriate model and to reuse it. The approaches of model selection emphasize the need for documentation and conceptual model that contains the modeling requirements. The reuse approach based on domain specific languages is widely used in system biology and dynamic system modeling. The lack of mathematical formalism to represent

PBM algorithms does not allow their extension. Therefore, the approach we followed to address the issue or PBM model component reuse includes the following requirements:

- find the right level of abstraction of model components to represent biophysical process regardless of the platform specificities;
- design a modeling language based on the shared concepts found in the biophysical processes;
- provide an automatic transformation that embed platforms specificities to generate source code, documentation and unit tests;
- keep the consistency between model specifications and implementation by providing an automatic transformation from PBM platforms to Crop2ML.

This approach will be designed and implemented in a multilanguage framework to support model components exchange between modeling frameworks.

Chapter 3. Crop2ML: An open-source multi-language modeling framework for the exchange and reuse of crop model components

Manuscript submitted for publication in *Environmental Modelling and Software.*

Midingoyi CA, Pradal C, Athanasiadis IN, Donatelli M, Enders A, Fumagalli D, Garcia F, Holzworth D, Hoogenboom G, Porter C, Raynal H, Thorburn P, Martre P. Crop2ML: An open-source multi-language modeling framework for the exchange and reuse of crop model components.

Abstract

Process-based crop simulation models are popular tools to analyze and predict the response of agricultural systems to climatic, agronomic, or genetic factors. They are often developed in crop simulation platforms to ensure their future extension and to couple different crop models with a soil model and a crop management event scheduler. The intercomparison and improvement of crop models are difficult due to the lack of efficient methods for exchanging biophysical processes between platforms. To overcome this limitation, we developed Crop2ML, a modeling framework that enables the description and the assembly (composition) of crop model components independently of the formalism of simulation platforms and the exchange of components between them. Crop2ML is based on a declarative architecture of modular model representation with an intermediate modeling language to describe biophysical processes and their transformation to crop simulation platforms. Here, we present the main components of the Crop2ML framework. Then, we describe the mechanisms of import and export between Crop2ML and simulation platforms. Finally, we discuss our approach and present some perspectives.

1. Introduction

The wide range of crop process-based models (PBM) reflects the evolution of our knowledge of the soil-plant-atmosphere system and their rich historical development for more than six decades (see reviewed of Jones et al. 2017; Muller and Martre 2019). The high diversity of PBM is due to their multiple applications and the complexity of the system influenced by several factors, e.g. weather, soil, crop management (Basso et al., 2013) and genotypic factors (Wang et al., 2019). Most of the PBM are continuous models, formalized using ordinary differential equations, but are implemented as discrete time simulation models using finite difference equations. They are commonly decomposed into simpler biophysical functions (e.g. phenology, morphogenesis, resource acquisition, pests and diseases impact) often implemented by recurrent equations with control flows Another common characteristic is that PBM simulate plant growth and development at the scale of the canopy (square meter of ground) or average plant level without spatial pattern with a daily or sub-daily time step.

PBM are often implemented in modeling and simulation platforms at a higher level of abstraction to facilitate model development (Rizzoli et al., 2008). These platforms offer not only scalable, modular, and robust modelling solutions but also the ability to analyze, evaluate, reuse and combine models. The diversity of PBM led the crop modeling community to compare their performance and to improve them by aggregating modelers' knowledge or by introducing improvements provided from diverse research groups under the umbrella of large international collaborative projects such as the Agricultural Model Intercomparison and Improvement project (AgMIP; Rosenzweig *et al.,* 2013). Studies conducted in the context of model inter-comparison and improvement exercises (e.g. Asseng et al. 2013; Wang *et al.,* 2017) pointed out the large uncertainty of PBM outputs and have analyzed the sources of uncertainty or the processes involved (Muller & Martre, 2019). These inter-comparison results show the potential and limits of PBM and highlight the need to analyze models at the process level, but also to exchange model components describing specific processes between simulation platforms (e.g. Wang *et al.*, 2017). The uncertainty of a PBM component may be related to its validity domain, inputs, parameters, structure, and the underlying scientific hypotheses (Walker et al., 2003). Epistemic uncertainty may arise from incomplete or lack of knowledge of these sources. The uncertainty of PBM results from the aggregation of the uncertainty of each of its component (Refsgaard et al., 2007). A framework that would allow the exchange of model components between different platforms would give crop modelers the ability to test alternative hypotheses in the same model, thus helping to reduce epistemic uncertainty.

Although most crop simulation platforms provide modular approaches and reuse techniques, there is little exchange of PBM components between them despite theoretical and application interests. PBM components often contain source code developed in different programming languages and are tightly coupled to the platforms. Therefore, model components are not seamlessly reusable outside the

modeling platforms in which they have been developed without recoding or wrapping them (Rizzoli *et al.*, 2008; Holzworth *et al.*, 2014a). Re-implementing a component in several platforms is a tedious and cumbersome task and requires a minimum knowledge of the different platforms. The wrapping solution treats components as black boxes taking little or no advantage of the framework (Rizzoli *et al.*, 2008) or as white boxes but with a high-level of technicalities (Pradal et al., 2008). Other reuse approaches in environmental modeling have been explored. Declarative modeling can provide portability and facilitate integration between independent, uncoordinated models Athanasiadis and Villa (2013). However, model specifications often remain informal and it is not always possible to distinguish the specifications from the scientific content of a model (i.e. its algorithm) or the structure from the implementation. Moreover, the publication of PBM components in scientific journals does not provide sufficient description associated with the modeled processes, which is a fundamental criterion for reuse (Pradal et al., 2013). This raises the problem of reproducibility and reliability of scientific results that are strongly linked to the platforms in which the models have been implemented and tested (Cohen-Boulakia et al., 2017; Hinsen, 2016).

Visual domain-specific languages such as Simile (Muetzelfeldt and Massheder 2003) or Stella (B. M. Richmond, 1985), provide a rich graphical interface to build models but become difficult to use for complex models and require many widgets to represent graphically nested control flows. Multiscale modelling and simulation frameworks (Pradal *et al.*, 2015; Marshall-Colon *et al.*, 2017) propose model interface designs, which enable communication of multi-language components as black boxes components. Other declarative modelling languages are also used in the Systems Biology community who have developed declarative open standard such as SBML (Hucka et al., 2010), CELLML(Cuellar et al., 2003), or NEUROML (Le Franc et al., 2012) to describe biological models. However, PBM cannot be described by equations formalized in System Biology (e.g. partial differential equations, reaction equations).

An alternative to the problem of PBM components reuse between PBM platforms is the use of a centralized framework that enables the development of PBM components regardless of the modeling platforms (Fig. 1). We followed this approach and developed a modeling framework called Crop2ML (Crop Modelling Meta Language) that separates the specification and structure of a model component from its implementation. Given that the wrapping solution was excluded because of the lack of transparency and high maintenance cost and that Crop2ML does not aim at replacing existing modeling platforms or at simulating components, it was decided to create a solution that generates components, from a metalanguage, for specific PBM platforms. It provides a centralized PBM components repository to store model components in a standard format to facilitate their access and reuse. This reuse approach is supported by the Agricultural Modeling Exchange Initiative (AMEI), which brings together some of the most widely used crop modelling and simulation platforms, including APSIM (Holzworth et al.,

2018), BioMA (Donatelli et al., 2010), DSSAT (Hoogenboom et al., 2019), OpenAlea (Pradal *et al.*, 2015), RECORD (Jacques Eric Bergez et al., 2016), Simplace (Enders et al., 2010) and other crop models such as STICS (Brisson et al., 2010) or *SiriusQuality* (Martre et al., 2006). Here, we first present the main components of Crop2ML framework. Then we describe the mechanisms of importing and exporting between Crop2ML and PBM platforms. Finally, we discuss our approach and present some perspectives.

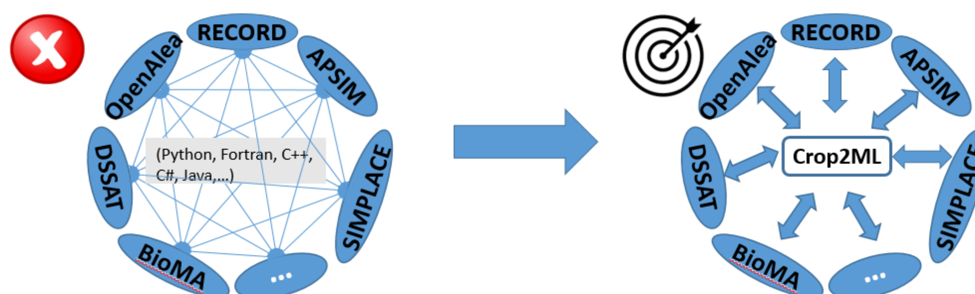

Figure 1: From a combinatorial to a centralized exchange framework. The schema illustrates the reduction of import export links between platforms in a centralized (right) versus combinatorial exchange framework.

2. Crop2ML: a centralized framework for crop model components development and sharing

Crop2ML is a framework for crop model components development, exchange and reuse between PBM platforms. It is designed following FAIR principles for research software (Lamprecht et al., 2019) to provide:

- *Simplicity*: Model specifications are defined using a declarative language (XML) with generic concepts shared between PBM platforms and model algorithms are encoded using a minimal language.
- *Transparency*: Models are shared as documented components in Crop2ML format.
- *Flexibility*: Model units are composed with a shared abstract representation of model structure.
- *Findability*: Model specifications include rich metadata and are assigned a globally unique and persistent identifier for each released version.
- *Reusability*: Model components are transformed into PBM platform compliant code to support efficient interoperability.
- *Reproducibility*: Model components can be executed and tested regardless of the PBM platforms.

- *Modularity*: Three levels of modularity of model are defined: *ModelUnit*, *ModelComposite* and package.
- We used the principles of Lamprecht et al. (2019) for assessing the FAIRness of Crop2ML framework (Table 1).

Table 1
FAIRness assessment of the Crop2ML framework following the principles of Lamprecht et al. (2019).

Principle	Description	Fulfilled	Comment
F1	Software and its associated metadata have a global, unique and persistent identifier for each released version	Yes	The Identifier of Crop2ML framework package is “Pycrop2ml” followed by its version 1.1.0. It can be found in Anaconda cloud. Version number uses the PEP-386 verlib conventions
F2	Software is described with rich metadata	Yes	Metadata covers the description, usage and accessibility of the package. Metadata in an Anaconda repository is specified in the meta.yaml file with a shared vocabulary for Anaconda project. The GitHub repository also contains description files but they do not use any controlled vocabulary.
F3	Metadata clearly and explicitly include identifiers for all the versions of the software it describes	Yes	All metadata include the version they apply to
F4	Software and its associated metadata are included in a searchable software registry	Yes	Anaconda cloud
A1	Software and its associated metadata are accessible by their identifier using a standardized communications protocol	Yes	Both software and metadata are accessible through HTTP/S: GitHub
A1.1	The protocol is open, free, and universally implementable	Yes	The protocol is open, free, and universally implementable
A1.2	The protocol allows for an authentication and authorization procedure, where necessary	Yes	Not necessary
A2	Software metadata are accessible, even when the software is no longer available	Yes	The GitHub repository contains DESCRIPTION files since version 0.99.6
I1	Software and its associated metadata use a formal, accessible, shared and broadly applicable language to facilitate machine readability and data exchange.	yes	The software is written in Python, a formal, machine readable and widely used language and Anaconda project metadata are available in an interoperable format (YAML) and GitHub DESCRIPTION files are written in machine readable reStructuredText
I2S.1	Software and its associated metadata are formally described using controlled vocabularies that follow the FAIR principles	Partially	Software: All Anaconda repositories use shared vocabularies for their description. Metadata: GitHub description does not use controlled vocabularies
I2S.2	Software use and produce data types and formats that are formally described using controlled vocabularies that follow the FAIR principles	Yes	Software uses model components in Crop2ML format and produces the source codes of the models in different languages and components conform to crop modeling frameworks. Crop2ML is also associated to a shared language used to describe model algorithm. This language can be translated to frameworks ‘languages.
I4S	Software dependencies are documented and mechanisms to access them exist	Yes	As stated in GitHub DESCRIPTION, Software dependencies areautomatically downloadable and accessible through Python package manager
R1	Software and its associated metadata are richly described with a plurality of accurate and relevant attributes	Yes	See comments for R1.1 and R1.2.
R1.1	Software and its associated metadata have independent, clear and accessible usage licenses compatible with the software dependencies	Yes	Software: under CeCILL-C license, well suited to the distribution of libraries and software reuse. Compatible with GNU GPL and specified in meta.yaml file. Metadata provided in GitHub description have a MIT license:
R1.2	Software metadata include detailed provenance, detail level should be community agreed	Yes	Commits on GitHub since version 0.99.
R1.3	Software metadata and documentation meet domain-relevant community standards	Partially	Developers provide documentation of model components but without following any community-agreed standard for doing that. Metadata do not follow any community standards.

2.1. Design and concepts of Crop2ML model specification

*S*oftware modularity is one of the main criteria of reuse. Jones *et al.,* (2001) propose key elements for modular model structure, which is an essential first step to enhance collaborative modelling effort. Crop2ML follows and extend these principals. In most PBM, the system is decomposed into compartments such as plant parts or soil layers that interact. For each compartment, different processes are described and assembled in components to simulate the behavior of the compartment. These processes can be subdivided into discrete, explanatory, independent biophysical sub-processes, which could be individually modeled (*ModelUnit*) and composed (*ModelComposite*). Modular model structure requires making an objective decomposition of the system to avoid coarse granularity models, which limit reusability. A *ModelUnit* should not encapsulate alternative assumptions and formalisms, making it easier to test them. In addition, the management of input and output data, such as data access, logging, and file generation, must be managed separately from the implementation of model component. These design principles foster the reuse of components, which are intended to be integrated and simulated with a large variety of input data formats in different PBM platforms. Moreover, to emphasis modularity, the temporal loop must be removed from modeling activity. This makes it possible to reuse the same activity with different modeling formalisms or simulation frameworks that manage temporal dynamics of the simulation differently.

Crop2ML provides a level of abstraction that enables a shared representation of model components between PBM platforms. A *ModelUnit* is defined with the following descriptive elements (Fig. 2a):

- a model description;
- a list of inputs;
- a list of outputs;
- an initialization of the state variables;
- a link pointing to the source of the model algorithm;
- a list of required mathematical functions (if required);
- a set of unit tests with shared parameterization.

A composite model includes the same elements as a *ModelUnit*, as well as the list *ModelUnits* that it contains and their links (Fig. 2b). However, in the case that the links between *ModelUnits* cannot be explicitly specified, an algorithm can be provided to specify how to evaluate outputs of this composite model.

Crop2ML model specification is based on the eXtensible Markup Language (XML). XML is a widely used declarative metalanguage for describing or structuring data in a portable format with some descriptive

elements. XML format is used in several PBM platforms for template parametrization and model simulation configuration (e.g. APSIM, BioMA, RECORD, Simplace, *SiriusQuality*). This reinforces our choice on this format since the transformation between different XML documents or in any language is relatively straightforward, allows using XML as a bridge between heterogeneous structures, and facilitating collaborative developments. Moreover, the use of XML and a formal description of model specifications and their associated metadata facilitate machine readability and model exchange. In the following sections, we describe the concepts of Crop2ML model specifications.

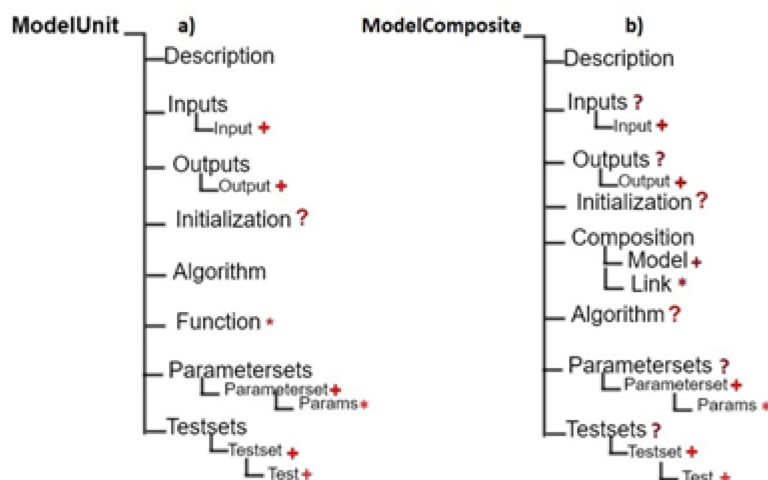

Figure 2: Crop2ML concepts for model specification. (a) ModelUnit. (b) ModelComposite. “+”, one or more elements; “*”, zero or more elements; “?”, zero or one element.

2.1.1. Description

The core description of a Crop2ML model contains the name of the model, an identifier that ensures the provenance of the model and a version number (Listing 1). The identifier of the model is specified to keep the property of the component. Since PBM are dynamic models, the time step is an important factor that is specified to allow a multi temporal-scale composition. In addition, other elements are described to provide rich metadata, including author names and affiliations, citable and findable references (e.g. doi) and a brief description of the model. The description also includes usage licenses compatible with the model dependencies.

```
<ModelUnit modelid="SQ.EnergyBalance.DiffusionLimitedEvaporation"
            name="DiffusionLimitedEvaporation"
            timestep="1"
            version="1.0">
    <Description>
        <Title>Diffusion Limited Evaporation Model</Title>
        <Authors>Pierre Martre</Authors>
        <Institution>INRA Montpellier</Institution>
        <Reference> Ritchie JT. 1972. Model for predicting evaporation from a row crop
                    with incomplete cover. DOI:10.1029/WR008i005p01204
                    Jamieson PD, Francis GS, Wilson DR, Martin RJ. 1995.
                    Effects of water deficits on evapotranspiration from barley.
                    DOI:10.1016/0168-1923(94)02214-5
        </Reference>
        <Abstract>the evaporation from the diffusion limited soil when the surface has
            dried sufficiently to provide a significant barrier to water vapor diffusion
        </Abstract>
        <URI> http://www1.clermont.inra.fr/siriusquality/?page_id=547 </URI>
        <License> MIT </License>
    </Description>
...
</ModelUnit>
```

Listing 1: Example of a Crop2ML ModelUnit description.

2.1.2. Inputs – Outputs

In Crop2ML, a component takes parameter and variable values as inputs and produces variable values as outputs. A variable is a quantity whose value may vary over time, while the value of a parameter does not change during model execution. Variables and parameters are distinguished with *input type* attribute and are categorized with *variable category* and *parameter category* attributes, respectively (Table 2).

Table 2
Category, definition, and example of variables and parameters in Crop2ML.

Input Type	Category	Definition	Example
Variable	State	Characterizes the behavior of a component	Leaf area index, weight of a plant part, canopy temperature
	Rate	Defines the change of one state variable	Transpiration rate, leaf growth rate
	Auxiliary	Intermediate variable computed by an auxiliary function	Dry matter partitioning, shoot number
	Exogenous	Driven variables that do not depend on other variables of the system or component	Mean air temperature, wind intensity
Parameter	Constant	Absolute constant	Boltzmann constant
	Soil	Soil parameter	N mineralization constant, maximum rootable soil depth
	Species	Crop parameter with fixed value for a species	Maximum respiration rate
	Genotypic	Crop parameter that can take different values for different genotypes (cultivars)	Phyllochron, grain filling duration

Crop2ML currently supports four basic types: integer, double, strings and logical. It also supports two collection types, lists and arrays, which contain a sequence of elements of basic types. They are explicitly specified in a *datatype* attribute. It also provides a common representation of date/time. The domain of validity of each variable is specified by *min* and *max* attributes. A measurement unit can also be associated to the variables and parameters. Listing 2 gives an example of inputs and outputs specifications.

```
<Inputs>
    <Input name="deficitOnTopLayers" description="deficit On TopLayers"
        variablecategory="auxiliary"  datatype="DOUBLE" default="5341"
        min="0" max="10000" unit="g/(m**2*d)" inputtype="variable"/>
    <Input name="soilDiffusionConstant" description="soil Diffusion Constant"
        parametercategory="soil" datatype="DOUBLE" default="4.2" min="0" max="10"
        unit="dimensionless" inputtype="parameter"/>
</Inputs>
<Outputs>
    <Output name="diffusionLimitedEvaporation"
            description="the evaporation from the diffusion limited soil "
            variablecategory="rate" datatype="DOUBLE" min="0" max="5000"
            unit="g/(m**2*d)"/>
</Outputs>
```

Listing 2: Example of inputs and outputs specifications of a Crop2ML model.

2.1.3. Initialization

State variables of a Crop2ML *ModelUnit* and *ModelComposite* are initialized at the start of a simulation and are specified with an *Initialization* element. This element is optional, and the default values of state variables are used if it is omitted. Initialization may also be a function that assigns initial values to state variables. In this case, the *Initialization* element contains the path to the source code of the initialization function.

2.1.4. Algorithm

Algorithm elements link the model specifications with the model algorithm. A model algorithm describes the behavior of a component in terms of a sequence of inputs, successive rules or actions, conditions and a flow of instructions from inputs to outputs including mathematical expressions. A model algorithm can be implemented in different programming languages. However, Crop2ML proposes to encode the model algorithm in a share language, CyML (Midingoyi et al. 2020b). The CyML source code is the common representation for model algorithm shared by the supported languages and platforms (see Section 2.2.).

```
<Algorithm  language="Cyml" filename="algo/pyx/diffusionlimitedevaporation.pyx" />
```

Listing 3: Example of link to an algorithm file.

2.1.5. Function

A function is a utility routine that can be called from the model algorithm or from other functions. It reduces the code length and improves the readability of the encoded algorithm. If a model needs an external function, this function must be declared in the model specification by referencing the path where the function is implemented. A function can also be used for model adaptations such as temporal aggregation or integration, unit conversion to link model components without changing their algorithms. Crop2ML provides a shared library of mathematical functions in different languages such as standard functions, interpolation, or upper and lower bound functions. Modelers can use these functions in their own algorithm, implemented in the CyML language.

2.1.6. Parameter sets and test sets

A Crop2ML model specification includes one or more sets of model parameterizations used for different unit tests (Listing 4). A parameterization is a set of values assigned to an input parameter of a model. It is described by a name and a description. A unit test in Crop2ML is described in the *Testsets* element and allows comparing estimated and expected outputs values. Several unit tests can be specified. They are described by their *name*, their *description* and the name of *parameters set* associated to them. Each test provides a list of values assigned to each variable and the expected values of the model outputs. A numerical precision could be associated with the *output of the test to check its validity.*

```
<Parametersets>
    <Parameterset name="set1" description="some values in there" >
        <Param name="soilDiffusionConstant">4.2</Param>
    </Parameterset>
</Parametersets>
<Testsets>
    <Testset name="first" parameterset = "set1" description="some values in there" >
        <Test name ="test1">
            <InputValue name="deficitOnTopLayers">5341</InputValue>
            <OutputValue name="diffusionLimitedEvaporation" precision ="3">6605.505</OutputValue>
        </Test>
    </Testset>
</Testsets>
```

Listing 4: Example of parameterization and unit tests in Crop2ML.

2.1.7. Model links

Model links are specified in a *ModelComposite* and depict how *ModelUnits* or *ModelComposites* are interconnected. A *ModelComposite* is a port graph (Andrei & Kirchner, 2009) that defines a dataflow where nodes are *ModelUnits,* and ports are inputs and outputs of the *ModelUnits*. Edges are oriented links connecting output ports of a source *ModelUnit* to the input ports of a target *ModelUnit* (Fig. 3). Three types of links must be specified: *InternalLink* is the connection between an input of one sub-model and the output of another sub-model, *InputLink* is the connection between an input port of a sub-model and an input port of the composite model, and *OutputLink* is the connection between a *ModelUnit* or *ModelComposite* output port, that can be either a *ModelUnit* or *ModelComposite*, and a *ModelComposite* output port. These connections show the hierarchical structure of a *ModelComposite*. This modeling approach enhances reusability and has been used historically with success (Wyatt, 1990).

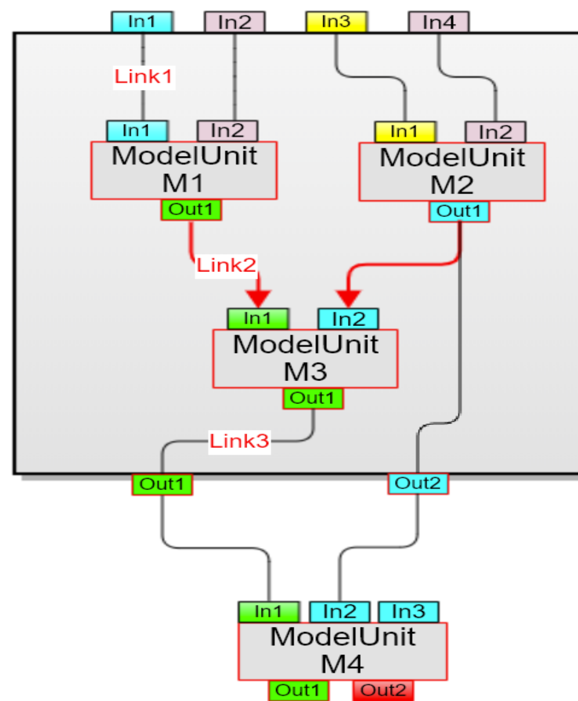

Figure 3: Graph of a *ModelComposite*. Three *ModelUnits* (M1 to M3) are connected to form a first level of composition, which is linked to a fourth *ModelUnit* (M4). Link1 is an InputLink, Link2 is an InternalLink, and Link3 is OutputLink.

2.2. CyML: the common modelling language of biophysical processes in crop models

We defined a set of common features resulting from the intersection of the programming languages supported by PBM platforms to propose a shared modelling language. A design choice was to define a subset of an existing language that can provide these common features. We needed a widely used high-level language with a low learning curve so that modelers with basic programming skills could efficiently use it. The transformation of a language with dynamic typing can make code transformation into programming languages with static typing ambiguous. Therefore, we choose Cython, a high-level language based on Python with explicit type declaration (Behnel et al., 2011). Cython is a language, which combines the expressive power of Python with the efficiency of C. It is compiled directly in efficient C code, which improves runtime speed and makes it possible to interact with C, C++ and Fortran source code. However, not all Cython syntax can be directly transformed in all target languages. For instance, the yield statement and anonymous functions are not supported by Fortran. Therefore, we defined CyML (Cython Meta Language), a sub-set of Cython to address the implementation of the model algorithm (Midingoyi *et al.*, 2020b).

We use CyML as a pivot language between various platform languages, which can be mapped to their syntax and semantics. The structure and syntax of CyML, as well as its transformation system to various languages and platforms is detailed in Midingoyi *et al.*, (2020b). In brief, CyML supports datatypes defined in the model specification and provides standard mathematical functions and operators. In addition to local variable declaration and assignment statements, control structures are used in the flow of instructions described by the encoded algorithms. These include conditional statements (if, elif and else) to check if a condition is satisfied before addressing part of an algorithm, sequential statement (for loop) with an incremental index on a data collection, and a repetitive statement (while) used to repeat part of an algorithm while a condition is satisfied. These structures can be nested. To support modular designs and the reuse of *ModelUnits* and functions, CyML provides import mechanisms, which assumes that imported *ModelUnits* or functions are referenced.

Crop2ML framework provides a source-to-source transformation system (CyMLT) which converts CyML source code in procedural (Fortran, Python, C++), object-oriented (Java, C#, C++, Python) and scripting or functional (R, Python) languages (Midingoyi *et al.*, 2020b). CyMLT implementation relies on the transformation of the abstract syntax tree (AST) generated from the syntax analysis of the CyML code. The AST is transformed to a self-contained representation of the source code called Abstract Semantic Graph, which is independent of the source language. CyMLT proposes a unique approach to transform the

Abstract Semantic Graph into readable source code in many different languages. The generated code is independent from the transformation system and can be run outside the Crop2ML framework. The transformation system integrates model documentation based on the model specification into generated code.

2.3. Crop2ML model package

In the context of large projects and collaborative work, it is useful to define some requirements or standards to have a common exchange way. Crop2ML provides a logical, standardized but flexible support to facilitate model sharing between modeling platforms through the definition of a directory structure (package template, Fig. 4). This template includes a folder that contains model description and associated algorithms, a repository of source code for each language and modeling platforms. It also includes a folder containing input data for a *ModelComposite* simulation, and a folder containing the unit tests. To save time and avoid forgetting mandatory files or folders during package creation, we created a cookiecutter (Roy, 2017) template that automatically generates Crop2ML package templates (https://crop2ml.readthedocs.io/en/latest/user/package.html). It increases model reusability by automatically generating a project that follows shared guidelines. Any *ModelUnit* or *ModelComposite* can be extracted as a stand-alone model from an existing package, tested, reused, or integrated in other *ModelComposite* or package. This notion of package-dependency increases the modularity of Crop2ML and avoid model duplicity.

2.4. Crop2ML model lifecycle management

Crop2ML aims at collaborative model development that support the entire model lifecycle, including model creation, edition, verification, validation, transformation, composition and documentation. Therefore, we developed tools and services to support all the steps of a Crop2ML model lifecycle.

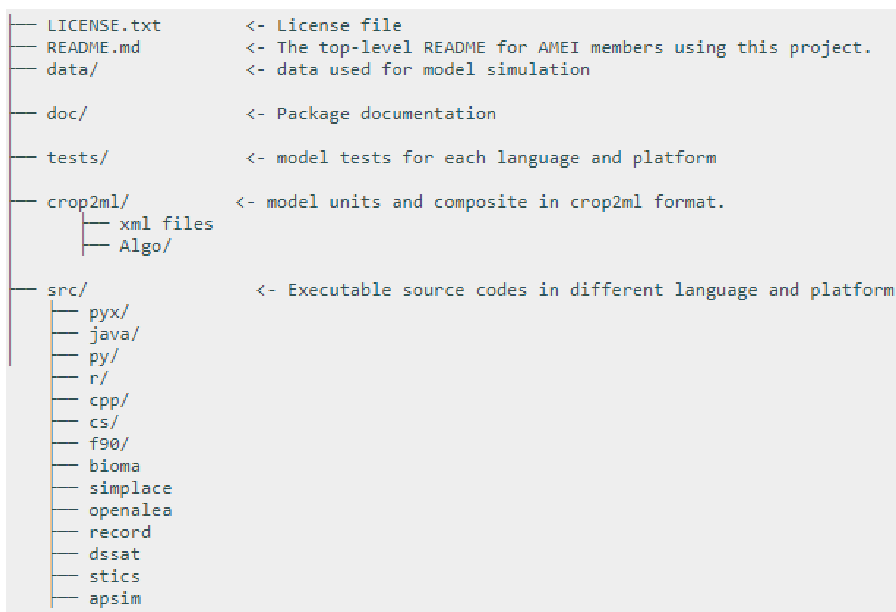

Figure 4: Tree view of the structure of a Crop2ML model component package.

2.4.1. Model analysis

Crop2ML models conform to a specific Document Type Definition (DTD) that describes Crop2ML concepts. Model analysis verifies if the model specifications are a well-formed XML document validated by Crop2ML DTD. The analysis of a *ModelComposite* consists of checking model composability through ports datatypes and units. Most XML editors can check the validity of an XML document against a DTD but the Crop2ML software environment (CropMStudio, see Section 3) ensures this.

2.4.2. Model validation

Crop2ML model components can be validated by executing unit tests. It consists of using the parameter and variable values from the model specification to produce unit tests in different languages. Unit tests are generated in Jupyter notebook format, a document format for publishing source codes and reproducible computational workflows that could be executed in the appropriate kernel in CropMStudio. This format is useful for code and documentation publishing and real-time collaboration when running on a remote server (Kluyver et al., 2016). Unit tests may also be associated with a model publication.

2.4.3. Model transformation

The success of Crop2ML model reuse through a white box approach comes from its ability to generate model components that conform to platform requirements. The transformation of a model component from a platform to another one goes through Crop2ML model representation. It relies on a system of transformation to and from Crop2ML and the platforms.

For some PBM platforms, meta-information of model components are described inside their implementation as documentation. For other platforms meta-information are encoded in a textual or visual programming language. CyMLT generates from Crop2ML model either appropriate documentation or variables and parameters specifications based on the artifacts of the target platforms. In addition, CyMLT generates model component algorithms in various languages. Given a model component provided by a platform, meta-information are extracted by identifying Crop2ML concepts inside the component to generate Crop2ML model meta-information. Moreover, algorithms in CyML are produced to obtain a complete Crop2ML model.

2.4.4. Model documentation

Sharing model knowledge requires detailed information on the model. Crop2ML generates model documentation from the model specification. From the relationships between the *ModelUnits* of a *ModelComposite*, the diagram flow of the *ModelComposite* is generated. It may constitute a part of the model documentation and gives a first description on the model component. This allows groups of modelers to discuss the model structure and evaluate the component.

3. Crop2ML software environment and tools

3.1. PyCrop2ML: A Python library for Crop2ML

Pycrop2ML is an open, modular, and extensible library developed in Python that implements all the steps of Crop2ML model lifecycle. It is designed to support the current Crop2ML model specifications but can easily be adapted to support future versions. Pycrop2ML can be integrated into other software projects as a plug-in. It allows:

- Verifying a Crop2ML model. This is ensured through a model parser based on the Crop2ML DTD.

- Transforming a Crop2ML *ModelUnit* to source code: PyCrop2ML integrates CyMLT that generates model components that conform to PBM platforms requirements.
- Transforming a CyML source code to various languages: Regardless of Crop2ML model specifications, any CyML source code can also be transformed into the target languages. This source code can be used as auxiliary functions for Crop2ML model development.
- Transforming source code to Jupyter notebook format: Each *ModelUnit* source code generated can be translated as a cell of Jupyter notebook, as well as, each unit test, allowing its execution in Crop2ML JupyterLab environment.
- Transforming a Crop2ML *ModelComposite*: A Crop2ML *ModelComposite* provided as a directed graph can be transformed to source code as a sequential order of the submodels.
- *Visualizing a ModelComposite*: Pycrop2ML provides a function to visualize a *ModelComposite* with the links between *ModelUnits* (Fig. 5).

PyCrop2ML is written in Python and can be executed via a command-line interface, inputting either a Crop2ML package or CyML source code, as well as, the target language or platform for transformation. Users with no knowledge of the Python language can easily run PyCrop2ML via the command line. The PyCrop2ML library incorporates three crop model components as model examples that can be used to test the different functionalities.

3.2. CropMStudio: A JupyterLab environment for Crop2ML model life cycle management

Crop2ML model specifications can be created or edited using any XML editor. However, to fulfil our objective of collaborative model development accessible to modelers with no specific programming skills, we developed a user-friendly interface based on the PyCrop2ML package to manage the lifecycle of Crop2ML model components (Fig. 6). Since Crop2ML models are transformed in different languages, it is useful to execute the unit tests in a single environment. Our solution, named CropMStudio, uses the JupyterLab environment (https://jupyterlab.readthedocs.io), an open source web application that allows working with code in different languages through different language backends kernels. We installed Python, Java, C#, C++, R and Fortran kernels to execute *ModelUnit* tests. The current version of CropMStudio can be accessed through a web browser and run locally like a desktop application. Another motivation to use JupyterLab is to make publication results reproducible in a shared environment based on the capacity to produce interactive and readable code documents (Kluyver et al., 2016).

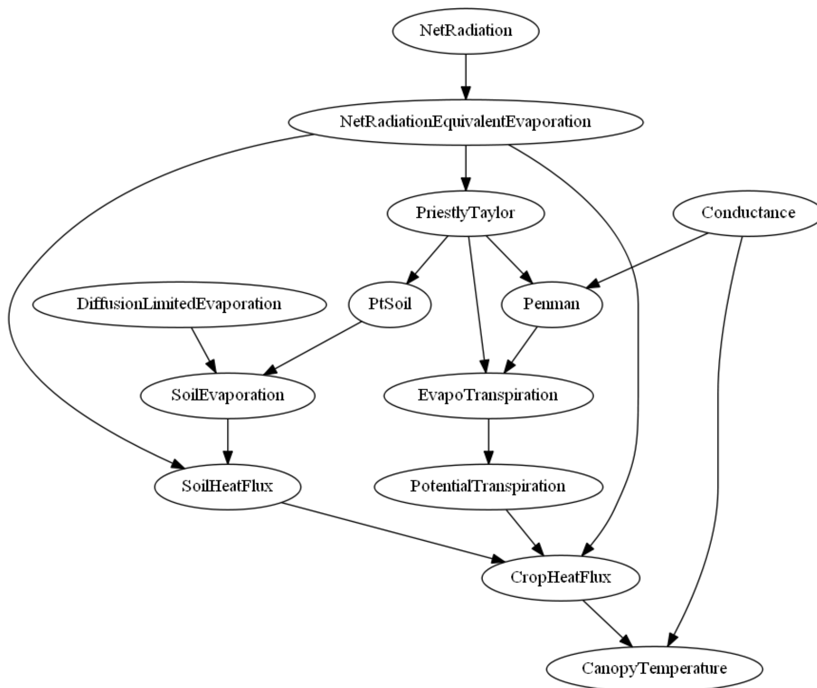

Figure. 5. Visualization of energy balance ModelComposite provided from SiriusQuality wheat model developed with the BioMA platform. Ellipses are ModelUnits and arrows represent the link between two ModelUnits

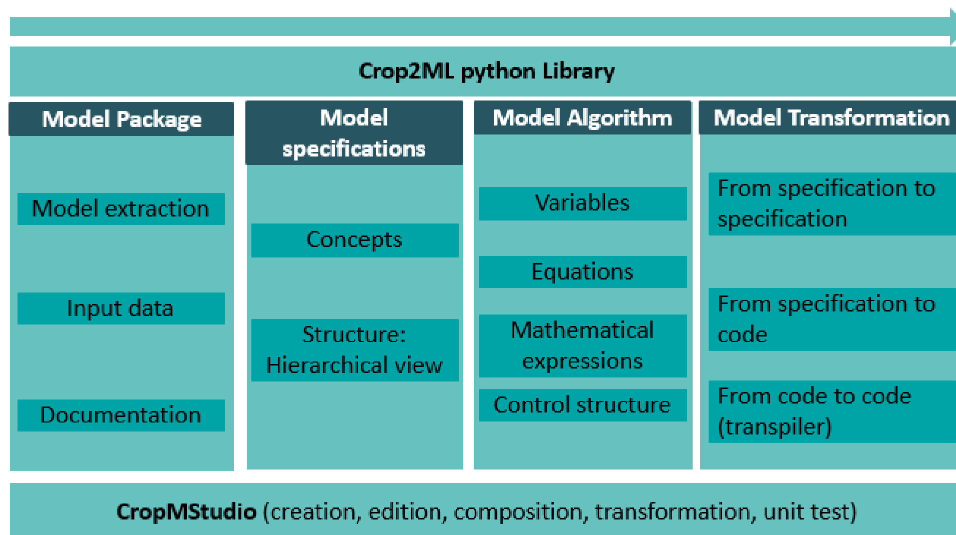

Figure 5: Schematic representation of the Crop2ML framework showing Crop2ML model lifecycle from the creation of a package to model transformation.

4. Interoperability between various simulation platforms

The interoperability between simulation platforms is based on two transformation processes (import and export) via Crop2ML. The import process consists of transforming any platform model component to Crop2ML model. The export process consists of transforming Crop2ML models to any platforms. Here we illustrate the interoperability of model components between five widely used PBM platforms with different architectures: BioMA, DSSAT, OpenAlea, RECORD, and SIMPLACE. These examples demonstrate that the concept of model exchange using Crop2ML is feasible and efficient using the Crop2ML protocol (Table 1).

4.1. BioMA

The Biophysical Models Applications (BioMA) as a follow up of the APES environment (Donatelli et al., 2010) is a software framework designed and developed by The Joint Research Center (JRC) of the European Commission (Donatelli and Rizzoli, 2008). It is used for running, calibrating, and improving modeling solutions based on biophysical models. Models supported by the BioMA framework are implemented in C# using the Object-Oriented paradigm. BioMA offers a modular and flexible architecture in three independent layers: (*i*) the model layer where fine-grained and composite models are implemented in components; (*ii*) the component layer where modeling solutions are developed from linked model components; and (*iii*) the configuration layer where the context (a purpose for what a modeling solution is defined) is set into the model to feed it with data and where adapters are provided to encapsulate legacy codes for reuse. The model layer is the one that is relevant as part of Crop2ML model exchange and can be well compared and mapped with the Crop2ML modeling approach.

At the model layer level, BioMA uses the strategy design pattern (Gamma et al., 1995) to make available a set of models that represent basic biophysical processes in a component through the same call and interface (IStrategy). Such models are called simple strategies, and they correspond to Crop2ML *ModelUnits*. Each simple strategy encapsulates model specification (input, output variables, algorithms), parameters and pre- and post-conditions tests. The component that embeds these simple strategies is called a 'composite strategy'. A composite strategy defines the model structure by invoking other strategies, and it is used to match the Crop2ML *ModelComposite* structure. The variables used as inputs and outputs of the strategies are stored as complex data types in specific classes, called 'domain classes'. The simple strategies of a component share the same domain classes. Usually, developers organize the variables of a component in

different domain classes according to the typology of the variables (e.g., all the variables related to the flux into a ‘rates’ Domain class, all the variables related to the state in a ‘states’ domain class). As Crop2ML focuses on the exchange of autonomous components instead of a modeling solution, all *ModelUnits* of a *ModelComposite* share the same context, and consequently, the corresponding strategies in BioMA will share the same domain class.

From Crop2ML to BioMA - The export to BioMA components is performed automatically by Pycrop2ML that allows generating simple strategy classes from Crop2ML *ModelUnits*, composite strategy class from Crop2ML *ModelComposite* and domain classes shared by strategies. After parsing all *ModelUnits*, different domain classes are generated according to the "*variablecategory*" attribute (state, rate, exogenous, auxiliary) of model inputs and outputs. They contain accessors methods of all variables of the proper category and constructors. A *VarInfo* class associated with domain classes is also generated and contains all properties of variables declared in the domain classes. The generated strategies implement all BioMa requirements, including the IStrategy interface, which contains the model algorithm, test pre- and post-conditions, and default values of the parameter set. The algorithms of the *ModelUnits* are translated into C# and incorporated in the Estimate method by using the CyML transpiler. Finally, the graph of models represented by the *ModelComposite* is converted to an ordered sequence of simple strategies calls. After the BioMA model components are generated, the domain classes and strategy classes are loaded into the BioMA Domain Class Coder and Strategy Class Coder, respectively. This step is used to prove that the generated files conform to BioMA requirements.

From BioMA to Crop2ML - The import process to Crop2ML allows retrieving inputs, parameters, and outputs of each Strategy class to obtain XML files of *ModelUnits* through a module based on the BioMA Model Component Explorer. Inputs and parameters of a strategy correspond to inputs in Crop2ML distinguished by the input type attribute. The model algorithm is manually translated into CyML. In the case where a composite strategy is described as a sequence of simple strategies calls, the graph of models is composed automatically. If the *estimate* method of a composite strategy incorporates some logic rules in the combination of strategies the model algorithm of the composite model is explicitly provided.

4.2. DSSAT

The Decision Support System for Agrotechnology Transfer (DSSAT, Jones *et al.*, 2003; Hoogenboom *et al.*, 2019) is an integrated crop modeling platform that incorporates crop simulation models for over 42 crops (as of Version 4.7) as well as tools to facilitate effective use of the models. At the core of DSSAT is

the Cropping System Model (CSM) designed in a modular format in which components are separated along scientific discipline lines and user interfaces to replace or add modules. Simulations are conducted at a daily time step or in some cases, at an hourly time step depending on the process and the crop model. State variables are updated at the end of each day of a simulation. The CSM is divided into modules including weather, management, soil, plant representing primary modules that contain secondary modules. The main program controls all the timing for the system, while a Land Unit module is used to control processing and data transfer between all primary modules. Each module is called to perform initialization of state variables at the beginning of a simulation, and at each iteration to calculate daily rates and perform daily integration of state variables.

Most of the secondary modules (e.g. soil water balance) are further subdivided into sub-modules for individual processes (e.g. snow accumulation). DSSAT sub-modules can be matched with Crop2ML *ModelUnits*, and DSSAT primary and secondary modules can be matched with Crop2ML *ModelComposite*. In DSSAT, model inputs and outputs are described inside a module and are used to create Crop2ML *ModelUnit* specifications. DSSAT modules have been developed in Fortran for performance reasons.

From Crop2ML to DSSAT – Export to DSSAT is performed through PyCrop2ML. It generates a submodule in Fortran 90 for each *ModelUnit*. It also generates a sequence of submodule calls for composite models. One issue is that Crop2ML does not manage the handling of input and output files. So, to integrate and execute the generated submodules in DSSAT, the modeler needs to manually add the input and output methods into the submodules.

From DSSAT to Crop2ML - All the steps of the import process to generate Crop2ML models from DSSAT components have been manually done. The information describing subroutines variables is not sufficient to produce Crop2ML models automatically. For each DSSAT secondary module, a transformation from Fortran to CyML is done after variable decomposition. In CSM, composite variables are used. For example, a weather type variable is defined to contain multiple pieces of information related to a single day of weather including day length, precipitation, maximum and minimum air temperature, and wind speed. In this case, local variables are extracted manually from the composite variable prior to accomplish the transformation. DSSAT relies on the use of the CMake utility to generate platform-dependent make files for CSM. It would be very useful when generating DSSAT components to edit the configuration file for CMake (CMakeList file.txt) in an automated way. This would simplify the manual addition of the generated components. Further streamlining of the process could be done by adding the capability to call the CyML transpiler directly from CMake.

4.3. OpenAlea

OpenAlea is a flexible and open-source component-based platform (Pradal *et al.*, 2008, 2015). It allows the decomposition of a system or application into separate and independent unit components and the assembly of these with other legacy components. It is used to implement efficiently plant models at different scales (cell, organ, plant, canopy) with heterogeneous data (raw data, digitized data, tree databases, 3D images). It is also designed to facilitate the interoperability between heterogeneous models and data structures from different scientific disciplines. It provides a visual programming environment called VisuAlea for the edition and composition of scientific models in a graphical user interface and to facilitate the access of different components and functionalities of the system. More than 30 different packages implemented in various languages from co-developers are available: Biophysics models, image processing, statistical analysis, L-systems (Fournier et al., 2010; Lindenmayer, 1968). The system architecture is based on the use of the Python language. The composition of models is represented by scientific workflows as directed acyclic graph. Different models of computation can be used to execute modeling solutions with different semantics (dataflow, discrete events, higher-order lambda dataflow). Components share the same plant representation, named the multiscale tree graph (MTG; Godin and Caraglio 1998). This formalism allows coupling different models at different spatial scales, while the model of computation can manage different time scales.

Although it is frequently used by the functional-structural plant modeling community, OpenAlea also offers development capabilities for crop models and presents a common modeling paradigm (component-based modeling) with Crop2ML. The concepts of *ModelUnit*, *ModelComposite*, and package in Crop2ML are equivalent to the concepts of Node, CompositeNode and Package in OpenAlea, respectively. In OpenAlea, a Node is the unit component and defines the granularity of a model. It is a callable object with typed inputs and outputs, which can be connected to other nodes to form a CompositeNode.

From Crop2ML to OpenAlea - OpenAlea allows the reuse of models implemented in different languages such as C, C++, Fortran, and Java but not in CyML. Therefore, to export a Crop2ML model to OpenAlea, PyCrop2ML is used to generate Python functions from each *ModelUnit* of a *ModelComposite*. We implemented a method to map Crop2ML data types to OpenAlea datatype interfaces so that each input and output has a well-defined interface that indicates their types and validity domain. This interface is also set to associate to each input, output, and model description a widget and automatically generate the GUI of each component based on an OpenAlea module integrated into PyCrop2ML package. These elements make it possible to construct a node (Fig. 6).

Thanks to the workflow structure of Crop2ML *ModelComposites*, OpenAlea CompositeNode are automatically generated and inputs/outputs compatibility is checked. This process includes the creation of Input and Output nodes from the InputLinks and OutputLinks of crop2ML *ModelComposite*, respectively. We developed specific functions to make the connection between Node input ports and CompositeNode inputs, Node output port and CompositeNode outputs, and to connect nodes within a CompositeNode. A CompositeNode is represented by a unique node reusable in another dataflow. During the import process, an OpenAlea package is generated that can then be opened in VisuAlea and managed by OpenAlea package manager. VisuAlea offers the possibility to access the models, their descriptions and the possibility to change their algorithms. This export process is performed with a loss of information because some attributes of Crop2ML model specification are not managed by OpenAlea, such as variable and parameter category and unit. To avoid this loss of information, all Crop2ML model attributes are included in the documentation of the Python functions associated to each node. In the future, the inputs/outputs interface of OpenAlea can be extended to take into account this information.

```
node.Factory(name='Model_name',
        authors='Model authors',
        description='Model description ',
        category='Unclassified',
        nodemodule='module file name',
        nodeclass='function name',
        inputs=[{'interface':Datatype(min='min',max='max'),'name':'name', 'value': 'default'}],
        outputs=[{'interface': Data type(min='min', max='max value'), 'name': 'output name'}],
        widgetmodule=None,
        widgetclass=None,
       )
```

Figure 6: Structure of a node in OpenAlea.

From OpenAlea to Crop2ML - The transformation from OpenAlea to Crop2ML uses the OpenAlea package manager integrated in PyCrop2ML. An OpenAlea *CompositeNode* (workflow) is automatically converted to Crop2ML *ModelComposite* (xml file). All nodes and the specification of the inputs and outputs of each node, such as their name, description, datatype, default value and validity domain are retrieved through the types of the interface to generate Crop2ML *ModelUnits* specifications. The python code associated with each node is manually translated into CyML to provide the algorithms of the model's specifications. The main limitation of the import process is that OpenAlea supports complex data structures that are not managed in Crop2ML and whose conversion is currently not supported.

4.4. RECORD

The REnovation and COORDination of agroecosystems modelling platform (RECORD; Bergez et al. 2013) aims at providing different services for building, simulating, and analyzing models in the context of agroecosystems. It uses the Virtual Laboratory Environment (VLE) simulation engine (Quesnel et al., 2009) , a generic modeling, simulation, and analysis environment based on the Discrete Event System Specification (DEVS) formalism (Zeigler et al., 2018). A graphical user interface (gvle) provides user-friendly tools to write model specifications, generate source code in C++, or execute models and analyze their output. Legacy software code may also be used in RECORD. This requires the user to either wrap the original source code (e.g. calling Fortran subroutines of STICS model from C++ Record libraries) or to modify the original code by removing temporal loops and managing inputs and outputs to be compatible to RECORD platform.

The concepts of *ModelUnit*, *ModelComposite*, and package in Crop2ML map to the concepts of atomic model, coupled model and package in RECORD, respectively. In RECORD, models (atomic or coupled) exchange information in the form of discrete events. As in Crop2ML, RECORD defines the package concept as an autonomous project facilitating model sharing and reuse. A C++ class is associated with the dynamic of each atomic model. However, the specification of an atomic model is embedded in the coupled model specification. Beside the difference equations formalism, RECORD provides other formalisms such as differential equations and decision rules commonly used in the agro-ecosystem context (J. E. Bergez et al., 2013).

From Crop2ML to RECORD - A RECORD coupled model relies on an XML file validated with a DTD. It specifies the atomic models that compose the coupled model and the way they are linked. This XML file called VPZ contains all the mandatory information for simulation. The export of a Crop2ML package to RECORD consists of generating the atomic model classes in C++ by using PyCrop2ML and a part of the vpz file showing the structure of the coupled model. Parameters are removed to obtain input ports of the RECORD coupled model defined in the Experiment section of the VPZ file. The Crop2ML *ModelComposite* links are mapped with those of RECORD to generate an acyclic models' graph. RECORD provides a tool to map the DTD of Crop2ML *ModelUnit* and RECORD atomic model to achieve the transformation.

From RECORD to Crop2ML - The transformation of a RECORD package to a Crop2ML package consists of parsing the VPZ file to generate the Crop2ML models specifications. However, this file is not sufficient to produce a complete Crop2ML model specification. For example, it does not provide the category of variables. The state variables are manually extracted from the model atomic classes. The

parameters are also extracted from the experimental conditions section of the VPZ file to build Crop2ML model specification. The main challenge of the import process is to generate an acyclic graph without retroaction loops from RECORD graph. The retroaction loop can be represented by the *InputLink* and *OutputLink* as the previous and current states are two distinct variables in Crop2ML. Thus, the InputLink comes from the composite input representing the previous state and the *OutputLink* connects with the composite output representing the current state. Finally, the *ModelUnit* algorithm results from the encoding of the compute method in CyML. A difference between Crop2ML and RECORD is that all the state variables are outputs in Crop2ML, while this is rarely the case in RECORD.

4.5. SIMPLACE

SIMPLACE (Scientific Impact assessment and Modelling Platform for Advanced Crop and Ecosystem management; Gaiser *et al.*, 2013) has been developed as a flexible modeling platform that attempts to meet the various demands of three user groups, scientists, engineers and decision makers, within one system only. This enables the different user groups to interact and bridge gaps. The platform operates with SimComponents as the smallest building blocks, which in most cases describe biophysical processes in the soil-plant-atmosphere system, which is described by combining several SimComponents through links established by input-output definitions. SimComponent maps to Crop2ML *ModelUnit*. GroupComponents combine SimComponents into logical structures of components that belong together, they map to Crop2ML *ModelComposite*.

With the graphical user interface view, non-experienced platform users like stakeholders, decision takers and students are able to run pre-defined model solutions and to analyze simulation results without further knowledge about the details of the underlying SimComponents or the model solutions. The XML based view is attached to the integrated development environment Eclipse. SimComponents algorithms are coded using object-oriented techniques in the programming language Java.

The Model Engine of the SIMPLACE platform is initialized using a model solution. It consists of constants, input data declaration, a model structure linking the SimComponents and an output description. It is defined by an XML DTD that also supports users in implementing and checking the semantic correctness of the modelling solution.

Model developers can implement their own modeling solutions with maximum flexibility using existing or their own SimComponents. The class structure of the Model Engine provides possibilities for model developers to extend the abstract SimComponent for implementing their own model descriptions or

SimComponent. It also contains an abstraction layer to support calibration, sensitivity analysis and regional application by extending generators, iterators and selectors according to the user requirements. Besides the flexibility and transparency of the open source implementation, SIMPLACE focuses on interaction between different modelling systems and widely enables the user to deep couple simulation runs using the various interfaces such as R, Matlab, Python, Octave. SIMPLACE additionally uses interfaces to import *ModelUnits* from APES, different FORTRAN based model implementations Lintul (Van Ittersum et al., 2003), HERMES (Kersebaum et al., 2019) and EPIC (Sharpley & Williams, 1990).

This implementation enables SIMPLACE to interact with the Crop2ML approach to import and export model components using the Crop2ML structures. SimComponents are exported to or imported from Crop2ML packages and GroupComponents are interfaced with Crop2ML *ModelComposite* structure. To further interact with the Crop2ML structure, the SIMPLACE XML-interface has been extended to include transfer maps to translate from SIMPLACE terms such as data types (e.g. DOUBLE or INTEGER), variable types (e.g. state, rate, or constant) to Crop2ML concepts.

In general, the interfacing with abstract languages like Crop2ML with their internal use of a (foreign) programming language consists in two steps that are mainly the same for import and export but have different order: (1) the transfer of algorithms and scripts from one programming language to another (from/to Java to/from Cython) and (2) the adaptation to the framework specific structures that are not language specific. An example for step (1) is the adaptation of a for loop. In the step (2) SIMPLACE explicitly sets values like Var.setValue(object) where CyML uses the simple definition like Var = object. The experience in importing and exporting structures via Crop2ML with SIMPLACE shows that huge parts of the process can be automatized. For ModelUnits that meet the following preconditions, import and export work without further adaption steps:

- Algorithms in the ModelUnit (in the SIMPLACE *process()* method) contain no language specifics.
- All class variables used in the ModelUnit are stateless within the component.
- No additional external code is used in the ModelUnit.

Many modelling units have been modified to meet these preconditions. However, there are still some complex SimComponents like SoilCNPK (Corbeels et al., 2005), SlimWater (Addiscott & Whitmore, 1991) that will require deeper changes in their structure to be transferred to the Crop2ML structure. In the future, it is planned to make entire modeling solutions transferable into a Crop2ML package. This would enable users to import complete simulation experiments seamlessly into other platforms. Some information specified in the SIMPLACE solution will then have to find its way into the Crop2ML structures like.

- Resource data structure including unique keys

- Transformers to adopt resource data structure to the model needs

Not readily included in the conversion process is the import and export of the existing unit tests to crosscheck implementations after import in other platforms.

Table 3

Import and Export process between Crop2ML and PBM platforms. A, automatic; P, partially automatic; M, manual.

PBM Platforms	From Crop2ML to PBM platforms				From PBM platforms to Crop2ML			
	A	P	M	To complete	A	P	M	To automatize
BioMA	✓					✓		Source transformation from C# to CyML
DSSAT		✓		Integration of I/O			✓	Source transformation from Fortran to CyML Remove I/O More variable description
Record		✓		Complete VPZ file		✓		Source transformation from C++ to CyML More variable description
OpenAlea	✓					✓		Source transformation from Python to CyML More variable description
SIMPLACE		✓		Correspondence between units (Crop2ML – SIMPLACE)		✓		Source transformation from Java to CyML

Table 3 summarizes the interoperability of model components between these platforms. Platforms are based on various programming languages, which requires the definition of transformation rules between CyML and various languages including C# (BioMA), Java (Simplace), C++ (Record), Python (OpenAlea) and Fortran (DSSAT) in both directions.

In order to illustrate Crop2ML concepts and transformation results, a phenology and an energy balance models are used. Phenology, the timing of crop development is the heart of most crop growth models and is an essential component of most crop modeling platforms. The energy balance model involves interconnected components that allows estimating canopy temperature, evapotranspiration, and heat transfer between the canopy and the air. These processes are implemented as BioMA standalone components (Manceau & Martre, 2018) of the wheat PBM *SiriusQuality* (He et al., 2012; Martre et al., 2006). The two

components were converted into Crop2ML packages, and then automatically translated into different languages and model components that conform to different PBM platforms. These packages are presented in Supporting Information Appendixes A and B. In Table 4 we illustrate how to represent a parameter and an algorithm in a Crop2ML Model Unit and its translation with CyMLT in Record, BioMA, and DSSAT. The implementations of the model differ between the platforms. For instance, DSSAT defines a subroutine with all the variables as argument, Record defines a class method (compute) with the variables as attributes of the class and uses specific operator "()" to manage temporal variables, while BioMA defines a class method (CalculateModel) that takes as argument data structures implementing each category of variables (state, rate, auxiliary, exogenous). The aim of model transformation is to provide to the platforms alternative model components that could easily replace their corresponding components to analyze the effects of new hypotheses into their modeling solutions.

The sequence of ModelUnits that compose a Crop2ML ModelComposite is formally modeled as a directed acyclic graph. This means that there is no feedback loop or retroaction at a given time step, instead they are usually represented by a cycle in the *ModelComposite*. Alternatively, a state variable can be defined explicitly as two variables with respect to the current and the previous time. Thus, a composite model may take as input a state variable at previous time and a state variable at current time as output, making implicitly a loop with respect to time advance. Another way to represent feedback inside a time step is to associate an explicit algorithm to the ModelComposite that defines how to run it. However, this feature is not supported by two simulation platforms (OpenAlea and RECORD)

Table 3. Declaration of the inputs and algorithm of a Crop2ML ModelUnit of the Penman-Monteith evapotranspiration model and the equivalent source code generated by CyMLT for Record, BioMA, and DSSAT. The declaration of a single variable is given as an example.

Framework or platform / language	Variable declaration	Algorithm
Crop2ML / CyML	<Input name="rhoDensityAir" description="Density of air" parametercategory="constant" datatype="DOUBLE" min="0" max="10000" default="1.225" unit="kg/m**3" inputtype="parameter"/>	def model_penman(float evapoTranspirationPriestlyTaylor=449.367, float hslope=0.584, float VPDair=2.19, float psychrometricConstant=0.66, float Alpha=1.5, float lambdaV=2.454, float rhoDensityAir=1.225, float specificHeatCapacityAir=0.00101, float conductance=598.685): """ """ cdef float evapoTranspirationPenman evapoTranspirationPenman = evapoTranspirationPriestlyTaylor / Alpha + 1000.0 * ((rhoDensityAir * specificHeatCapacityAir * VPDair * conductance) / (lambdaV * (hslope + psychrometricConstant))) return evapoTranspirationPenman

Record / C++	//Model parameters
	/**
	* @brief Density of air (kg/m**3)
	*/
	double rhoDensityAir;
	//...
		virtual void compute(const vd::Time& /*time*/)
		{
		evapoTranspirationPenman = evapoTranspirationPriestlyTaylor() / Alpha +
		(1000.0d * (rhoDensityAir * specificHeatCapacityAir * VPDair()
		* conductance() / (lambdaV * (hslope() + psychrometricConstant()))));
		}

BioMA / C#

```
VarInfo v4 = new VarInfo();
v4.DefaultValue = 1.225;
v4.Description = "Density of air";
v4.Id = 0;
v4.MaxValue = 10000;
v4.MinValue = 0;
v4.Name = "rhoDensityAir";
v4.Size = 1;
v4.Units = "kg/m**3";
```

```
private void  CalculateModel(SiriusQualityEnergybalance.EnergybalanceState s,
                             SiriusQualityEnergybalance.EnergybalanceState s1,
                             SiriusQualityEnergybalance.EnergybalanceRate r,
                             SiriusQualityEnergybalance.EnergybalanceAuxiliary a,
                             SiriusQualityEnergybalance.EnergybalanceExogenous ex)
{
    double evapoTranspirationPriestlyTaylor = r.evapoTranspirationPriestlyTaylor;
    double hslope = a.hslope;
    double VPDair = a.VPDair;
    double conductance = s.conductance;
    double evapoTranspirationPenman;
    evapoTranspirationPenman = evapoTranspirationPriestlyTaylor / Alpha +
                               (1000.0d * (rhoDensityAir * specificHeatCapacityAir * VPDair
                               * conductance / (lambdaV * (hslope + psychrometricConstant))));
    r.evapoTranspirationPenman = evapoTranspirationPenman;
}
```

DSSAT / Fortran

```
!          * name: rhoDensityAir
!                   ** description : Density of air
!                   ** datatype : DOUBLE
!                   ** min : 0
!                   ** max : 10000
!                   ** default : 1.225
!                   ** unit : kg/m**3
```

```
SUBROUTINE model_penman(evapoTranspirationPriestlyTaylor, ...)
    REAL, INTENT(IN) :: evapoTranspirationPriestlyTaylor
    ...
    evapoTranspirationPenman = evapoTranspirationPriestlyTaylor / Alpha +  &
            (1000.0 * (rhoDensityAir * specificHeatCapacityAir * VPDair *  &
            conductance / (lambdaV * (hslope + psychrometricConstant))))
END SUBROUTINE model_penman
```

5. Discussion

The Crop2ML framework enables the user to exchange and reuse biophysical components between various PBM platforms through shared declarative specifications. The use of a minimal language to describe the model algorithm once and the transformation system facilitates the model component reuse. *ModelUnits* and *ModelComposite* could be accessed and composed following a white box approach. Therefore, the Crop2ML approach greatly increases the ability of modelers to share their algorithms. The protocol will allow modelers to borrow components easily and will facilitate their intercomparison in different PBM platforms.

5.1. How does Crop2ML address model reuse with respect to other initiatives?

Some initiatives addressed model reuse by providing multi-scale and multi-language integrative frameworks such as *Crops in silico* (Marshall-Colon et al., 2017) or OpenMI (Buahin & Horsburgh, 2018). These frameworks can compose and simulate heterogeneous models provided by different frameworks through a communication interface. The model components are often wrapped and are represented as black-box components. All state variables are not always exposed as model outputs, which may limit their integration in an existing modeling solution. Therefore, these frameworks enhance model reuse in their own environment but they do not address reusability with other PBM platforms. Many existing PBM platforms do not support the coupling of models written in multiple languages (e.g. BioMA, APSIM next generation).

Donatelli and Rizzoli (2008) proposed a design pattern for platform-independent model components to enhance modularity and to facilitate model reuse in several PBM platforms via simple wrappers. However, this approach fixes the structure of the components. The lack of specification or meta-information makes the reuse of model components between platforms difficult. Even in component-based systems, explicit information about the component itself and its inputs and outputs (types, units and boundary conditions) are required to ensure a syntactic composability and to meet the specificities of the platforms. Moreover, the knowledge of the structure underlying the source code of a component is also required to systematically extract model information (variables and algorithms) for their transformation and integration in different platforms. We thus argue that model component reuse is improved if it is supported by model specification. Crop2ML defines an abstract representation of model design shared by PBM platforms through some shared concepts enriching or extending those proposed by Athanasiadis *et al.*, (2011) with other attributes and a

formal and shared description of unit tests. We included unit tests in Crop2ML specifications to ensure model transformation validation and some imperative constructs for model dynamics.

Several initiatives have used declarative modeling to describe model specifications and address model reuse issues. The approach proposed by Villa (2001) is similar to ours but it is limited to models where the dynamics of the modeled processes is represented by simple mathematical expressions without control structures, which does not match crop-modeling context. Hucka *et al.*, (2003) used MathML (Ausbrooks et al., 2003) to express interactions between variables through mathematical formalisms well defined in the systems biology community. This approach is similar to that of Rizzoli *et al.* (2008) and is interesting when processes are governed by differential equations. However, in the PBM context, simulation platforms use algorithms to describe processes rather than mathematical formalisms with differential equations. Moreover, in PBM, variables that drive the system are temporal series that change the behavior of the system at discrete time. This does not require finding a general solution of recurrent equations used in crop models but rather estimating at each time step the state variables of the system.

Automated model transformation is a core aspect of model-driven development (Cuadrado & Molina, 2007). It uses Model-Driven Engineering (MDE) principles based on metamodeling concepts. Crop2ML is in line with MDE. It defines structured concepts representing its metamodel, with which all Crop2ML models are conform, and a model transformation to generate PBM platforms' components. Model Driven Architecture (A. W. Brown, 2004) is a framework of MDE that provides several standards languages (e.g., ATL, QVT, ETL, Henshin, VIATRA, and Stratego) for model transformation (Jouault & Kurtev, 2006; Kurtev et al., 2006). Crop2ML is based on a transformation process through a set of refinement of models and code with some extensible rules defined as templates in Python. Most MDE approaches allow model to model or model to code transformation where a model represents the specification in our case. However, the use of transformation language standards was inappropriate in our context to unify transformation process towards many languages with different paradigms (Bucchiarone et al., 2020). Crop2ML produces code in a target language but also adapt the code to fit with PBM platform specificities. To our knowledge, model transformation languages in MDE do not support code generation in multiple languages with extended features in the same environment.

5.2. Connecting Crop2ML to PBM platforms

Given that Crop2ML datatypes do not handle complex structures other than arrays and lists, some compromises or transformation should be made to the import-export process on platform side with respect

to handling other data structures used in platforms. As an example, BioMA provides the Dictionary data type that is a mapping between keys and values to represent either input or output variables. This data type is not shared by PBM platforms and not handled in Crop2ML. As an alternative, Dictionaries can be expressed in Crop2ML as two list datatype variables that represent keys and values of the dictionary.

The simulation algorithm defining the feedback loop is explicitly described as control flow in some platforms (e.g. BioMA) but this is not the case in other platforms (e.g. Record, where the VPZ file representing the simulation model file is handled by the simulation engine VLE). Different simulation engines are based on different models of computation used by the platforms such as dataflow (e.g. OpenAlea), DEVS simulation (e.g. Record), control flow (e.g. BioMA, DSSAT, and Simplace). These models of computation are used to coordinate the execution of the model. The current version of Crop2ML framework does not take into account the specificities of simulation engines and addresses components which can be sequentially composed.

The Crop2ML transformation system is designed to support the specificities of the target PBM platforms. However, the semantic of a Crop2ML model is based on shared concepts to describe at a high level a biophysical process designed as a discrete-time model. There is no semantic reason to support the description of each instance of the concepts. For example, since we have not defined a convention to name process variables, the integration of a Crop2ML component into a PBM modeling solution requires adapting the name of its variables. In the future, we could annotate Crop2ML models to add semantic information to make semantic links between any Crop2ML model variables or parameters with those of model components of PBM platforms. This will also allow a semantic composability of Crop2ML models instead of a syntactic composability that analyzes whether the pair of variables to be linked are compatible. However, this would require the crop modeling community to agree on shared semantics and ontologies of crop model variable and parameter representations. In addition to facilitating the exchange and reuse of model components, semantic descriptions of model variables and parameters would facilitate the linking of crop models to plant phenomics data (Neveu et al., 2018).

The import process into Crop2ML is more mixed regarding the overall difference between PBM platforms. It is much easier to start with concepts shared and reused by PBM platforms than to start from divergent views of model representations to achieve a particular result. Some PBM platforms need to extend their concepts for model specification or to provide a rich model documentation in order to produce complete Crop2ML model specifications. This reveals the need of a good level of abstraction to represent a model in various PBM platforms. The higher the level of abstraction, the further the description moves away from the platforms and the less easy it is to understand. If the level of abstraction is low, it is not always possible to represent all features of the models present in the platforms.

5.3. Future developments

A common model repository infrastructure is essential for efficient model exchange (Le Novere, 2006). Currently, Crop2ML model components are stored in Github repositories. We need to provide a Crop2ML model repository to store models in a shared format to make them easily accessible and reusable by the PBM community. This repository should aim at hosting alternative biophysical processes. It will help modelers to operate on multiple model components, compare processes, or evaluate the impact of the integration of alternative models of biophysical processes in crop models. The success of the Crop2ML repository requires that the PBM community gives access to their models by feeding the repository which will be curated by AMEI consortium to avoid error propagation.

Crop2ML has some limitations, which can be addressed in the next versions either by extending the model specifications with shared concepts or by adapting the target PBM platforms to Crop2ML specification and language. It is an ongoing, long-term work, to satisfy platform requirements and facilitate Crop2ML model life-cycle management to make Crop2ML a standard for the crop modeling community.

The transformation of a model component of a PBM platform into a Crop2ML package requires to rewrite the model algorithms in the CyML language. This limit will be addressed by extending the CyML transpiler to a bidirectional transpiler. Thereby, PBM platforms could provide model algorithms in the language they use and the extended CyML transpiler will transform them in CyML and target languages used by other PBM platforms (Fig. 7). This is a two-step process. First, the model algorithms in the language of the source PBM platform will be parsed and an AST will be generated. Second, the rules for transforming this AST into the CyML AST will be applied. The second step will reuse the CyML transformation tool developed by Midingoyi *et al.*, (2020b) to produce model algorithms compatible with other languages and PBM platforms.

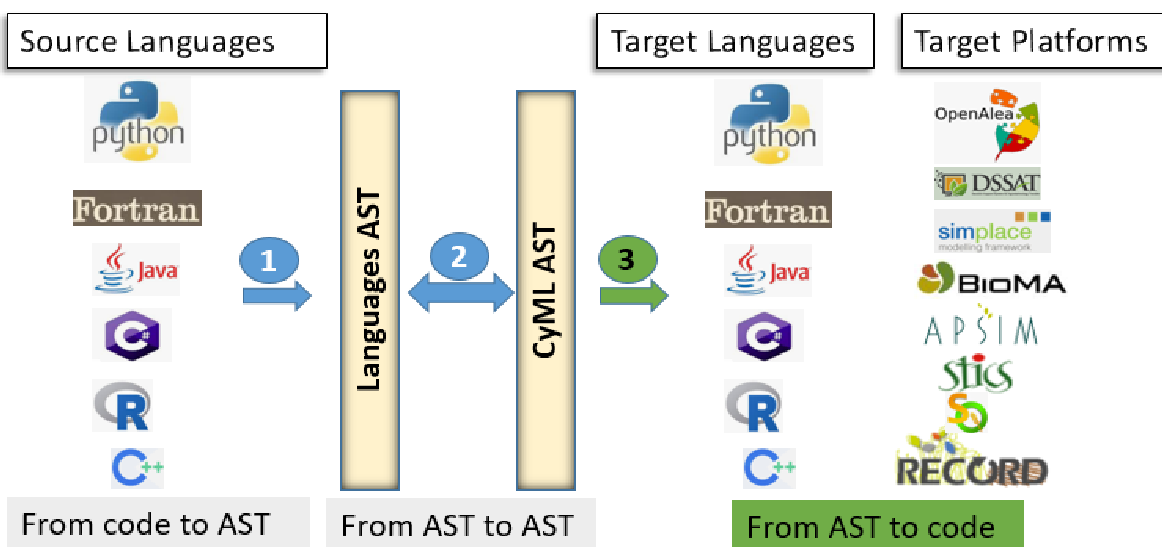

Figure 7: Schema illustrating CyML transformation extensibility to support bidirectional source transformation.

Other future developments of Crop2ML include:

- enhance Crop2ML model repositories with model annotation to link publications to models for reproducibility;
- add unit checks and conversions in Crop2ML to improve model validity;
- define a methodology to link Crop2ML with plant structure representation for multiscale viewing and analysis;
- Support for an ontology to allow better Crop2ML model interpretation and improve transformation between PBM platforms.
- Extend Crop2MLab prototype: The current prototype allows managing Crop2ML models and transforming them into target languages and platforms. Future developments will include bidirectional transformation and the creation of a web interface on a remote server in order to give users the possibility to handle Crop2ML model lifecycle without local installation.

6. Conclusion

At the boundary between modeling and software engineering, this paper addresses crop model component reuse by proposing Crop2ML. Despite all the differences between crop models development and simulation platforms, some common features were found that enabled model representation regardless of the PBM platform specificities. Crop2ML provides some structured concepts to support the definition of *ModelUnit* and *ModelComposite* and allows their transformation to make them compatible with PBM platforms at implementation level. Therefore, Crop2ML defines a new unified crop model representation that considers the abstraction of PBM component features in several PBM platforms. Moreover, Crop2ML uses a domain specific language to describe biophysical processes and auxiliary functions to represent model dynamics based on a subset of the Cython language, which can then be automatically transformed into different target languages. Crop2ML proposes an open framework to manage all the steps of model lifecycle.

Supporting Information

Appendix A. Crop2ML Energy balance component

https://doi.org/10.5281/zenodo.4292231

Appendix B. Crop2ML Phenology component

https://doi.org/10.5281/zenodo.4292245

Chapter 4. Reuse of process-based models: automatic transformation into many programming languages and simulation platforms

Published in *in silico Plants*.

Cite as:

Midingoyi CA, Pradal C, Athanasiadis IN, Donatelli M, Enders A, Fumagalli D, Garcia F, Holzworth D, Hoogenboom G, Porter C, Raynal H, Thorburn P, Martre P (2020) Reuse of process-based models: Automatic transformation into many programming languages and simulation platforms. *in silico Plants,* diaa007. https://doi.org/10.1093/insilicoplants/diaa007

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.

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.* 2014b). 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.* 2020a) 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 (Cuellar et al., 2003; Gleeson et al., 2010; Hucka et al., 2003). 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 (Fernique & Pradal, 2018; Plaisted, 2013). 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 & 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 sets of parameter 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 to 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 is 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., 2011). 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.

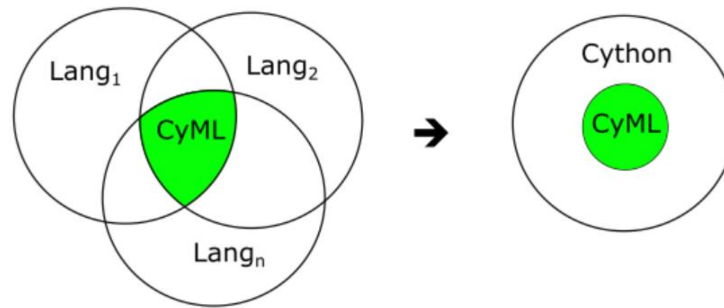

Figure 1: From the intersection of a set of languages features to a definition of an abstract language CyML, defined as a subset of Cython. $Lang_i$ corresponds to a minimal language supported by a crop simulation platform "i". The number of circles (n) in the left corresponds to the number of platforms.

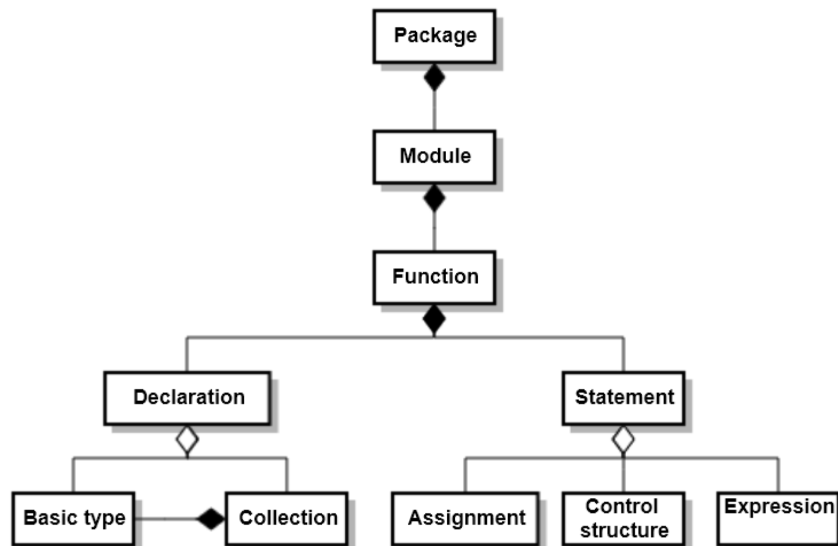

Figure 2: Main concepts supported by the CyML language. Black diamonds indicate composition ("contains") relationships and white diamonds indicate a specialization ("is-a").

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. "="). Therefore, an assignment statement can 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 of 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.

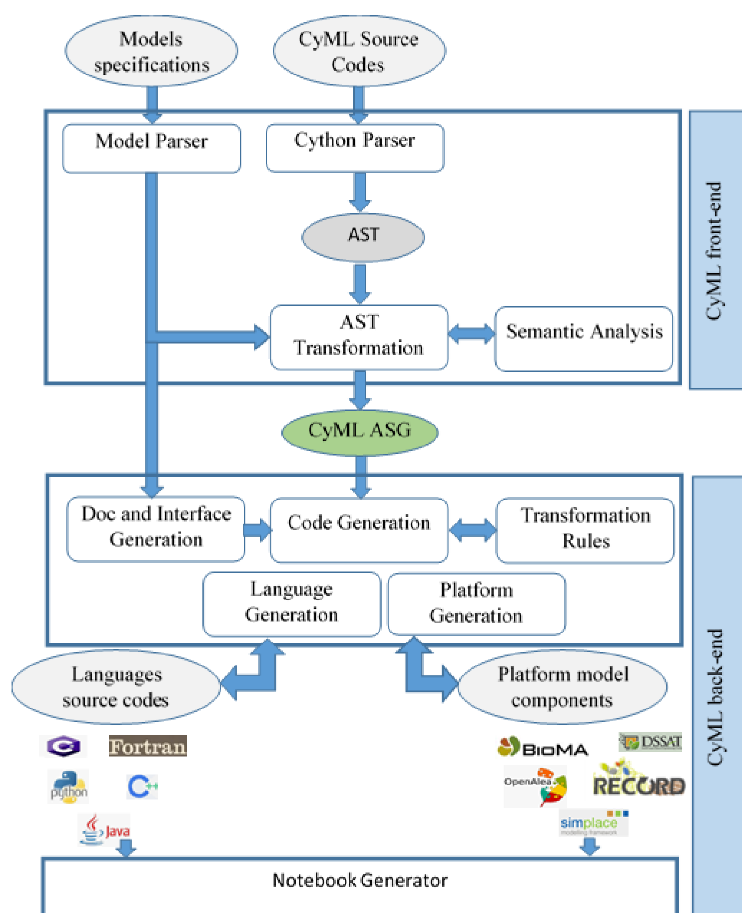

Figure 3: Design architecture of the one-to-many CyML transformer (CyMLT). It takes as input a model unit algorithm implemented in CyML with associated model specifications and applies a transformation workflow to produce crop model components or source code in different languages for different platforms.

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 *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.

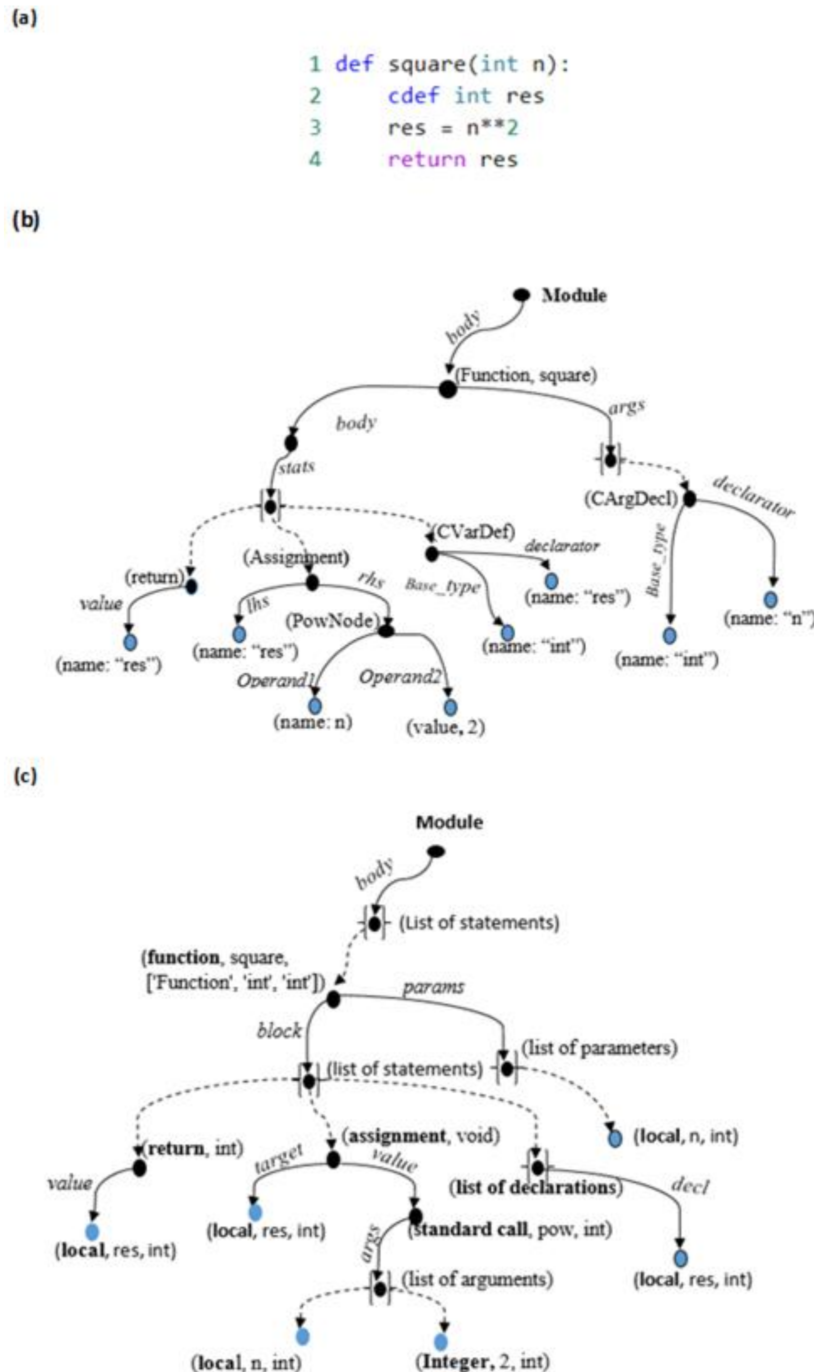

Figure 4: Example of abstract syntax tree (AST) and abstract semantic graph (ASG). (a) definition of function "square" in CyML. (b) simplified view of AST of function "square" where the internal nodes in black represent Cython constructs and the final node in blue a variable or constant. (c) Simplified view of ASG with 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 $(p\ c)$.

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.

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.

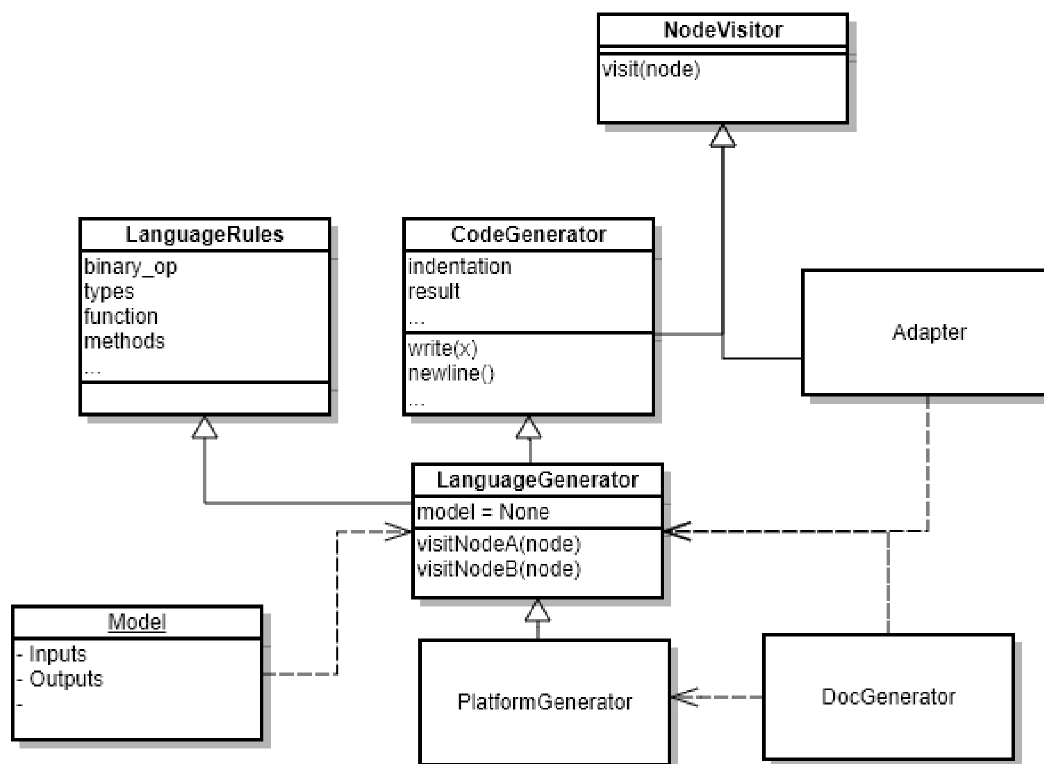

Figure 5: Class diagram illustrating the implementation of the one-to-many CyML transformer (CyMLT).

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 (Supporting Information Listing S2).

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.

Java

```java
1   public class Shootnumber
2   {
3       private double sowingDensity;
4       public double getsowingDensity()
5       { return sowingDensity; }
6       public void setsowingDensity(double _sowingDensity)
7       { this.sowingDensity = _sowingDensity; }
8       ...
9
10      public Shootnumber() {...}
11      public void  Calculate_shootnumber(...)
12      {
13          ...
14      }
15      public static int fibonacci(int n)
16      {
17          ...
18      }
19  }
```

C#

```csharp
1   public class Shootnumber
2   {
3       private double _sowingDensity;
4       public double sowingDensity
5       {
6           get { return this._sowingDensity; }
7           set { this._sowingDensity = value; }
8       }
9       ...
10      public Shootnumber() {...}
11      public void  Calculate_shootnumber(...)
12      {
13          ...
14      }
15      public static int fibonacci(int n)
16      {
17          ...
18      }
19  }
```

Fortran

```fortran
1       SUBROUTINE model_shootnumber(
2           sowingDensity, &
3           ...)
4           IMPLICIT NONE
5           REAL, INTENT(IN) :: sowingDensity
6           ...
7       END SUBROUTINE model_shootnumber
8
9       RECURSIVE FUNCTION fibonacci(n) RESULT(res_cyml)
10          IMPLICIT NONE
11          INTEGER, INTENT(IN) :: n
12          ...
13      END FUNCTION fibonacci
```

C++

```cpp
1   class Shootnumber
2   {
3       private:
4           float sowingDensity;
5           ...
6       public:
7           Shootnumber();
8           void  Calculate_Model(...);
9           int fibonacci(int n);
10          float getsowingDensity();
11          void setsowingDensity(float _sowingDensity);
12          ...
13  };
```

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

```java
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**Erreur ! Source du renvoi introuvable.** 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 a 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

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

Python

```
1  def sum_(x):
2      y = 0
3      for i in x:
4          y = y + i
5      return y
```

C++

```
1   template<size_t SIZE_0>
2   int sum_(const array<int, SIZE_0>& x)
3   {
4       int i;
5       int y = 0;
6       for(const auto& i_cyml : x)
7       {
8           i = i_cyml;
9           y = y + i;
10      }
11      return y;
12  }
```

C#

```
1   public class Test
2   {
3       public static int sum_(int[] x)
4       {
5           int i;
6           int y = 0;
7           foreach(int i_cyml in x)
8           {
9               i = i_cyml;
10              y = y + i;
11          }
12          return y;
13      }
14  }
```

Java

```
1   public class Test
2   {
3       public static int sum_(Integer [] x)
4       {
5           int i;
6           int y = 0;
7           for(Integer i_cyml : x)
8           {
9               i = i_cyml;
10              y = y + i;
11          }
12          return y;
13      }
14  }
```

Fortran

```
1   FUNCTION sum_(x) RESULT(y)
2       IMPLICIT NONE
3       INTEGER , DIMENSION(: ), INTENT(IN) :: x
4       INTEGER:: y
5       INTEGER:: i
6       INTEGER:: i_cyml0
7       y = 0
8       DO i_cyml0 = 1, SIZE(x)
9           i = x(i_cyml0)
10          y = y + i
11      END DO
12  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) and *DiscreteTimeDyn* interface, respectively. Another approach followed by platforms is component-based programming. A model developer creates a component that inherits from 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.

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.

```
1  public class Shootnumber extends FWSimComponent
2  {
3
4      // Constructor
5      public Shootnumber(){
6          super();
7      }
8
9      @Override
10     protected void process()
11     {
12         // model algorithm
13     }
14
15     @Override
16     protected void init()
17     {
18         // Component initialization
19     }
20
21     @Override
22     protected FWSimComponent clone(FWSimVarMap aVarMap)
23     {
24         // creates a clone from this Component for use in other threads
25     }
26
27 }
```

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 conforming to platform requirement.

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 model 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.

```
1    binary_op = {"and": "&&",
2                 "or": "||",
3                 "not": "!",
4                 "%":"%%",
5                 ...
6                 }
7
8
```

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.

```
1     functions = {
2         'math': {
3             'ln':          'log',
4             'log':         'log',
5             'tan':         'tan',
6             ...
7         },
8        'io': {
9             'print':    translatePrint,
10             ...
11         },
12         'system': {
13             'min': 'min',
14             'modulo': translateModulo,
15             ...
16         }
17     }
18
```

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).

```
def visit_bool(self, node):
    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.

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

Listing 16: Implementation of standard function call.

This implementation approach is followed for all types of nodes, and it 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.* 2014b) 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 are 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 platform specificities. A model component can be reused, improved, integrated and simulated in various platforms. This improves the diffusion of models, sharing them as software and scientific artifacts, and thus, enhancing transparency and reproducibility of crop models. Moreover, with CyML, 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 the same 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, Nunnari and Heloir (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, parameter values and write output values in a file are 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.* 2014b). 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 alternative 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 sets up various analyses 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 is available publicly on Github at https://github.com/AgriculturalModelExchangeInitiative/PyCrop2ML. Full documentation for CyML and CYMLT can be found at https://pycrop2ml.readthedocs.io.

Supporting Information

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

Table S1. Mapping of basic data types between CyML and the languages supported by CyMLT.

Table S2. Mapping of arithmetic operators between CyML and the languages supported by CyMLT.

Table S3. Precedence pecking order in CyML language and the languages currently supported by CyMLT.

Table S4. Mapping of built-in functions between CyML and the languages supported by CyMLT.

Table S5. Mapping of flow control statements between CyML and the languages supported by CyMLT.

Listing S1. A Crop2ML model specification for the shoot number model

Listing S2. CyML code of the shootnumber unit model of the WheatPhenology composite model.

Table S1

Mapping of basic data types between CyML and the languages supported by CyMLT.

Data type	**Language**						
Languages	**CyML**	**Python**	**C#**	**Java**	**F90**	**R**	**C++**
integer	int	int	integer	integer	Integer::	numeric	int
real	float	float	double	double	Real::	numeric	float
character	str	str	string	string	Character::	character	string
boolean	bool	bool	boolean	boolean	Boolean::	logical	bool

Table S2

Mapping of arithmetic operators between CyML and the languages supported by CyMLT.

	Language						
Operator	**CyML**	**Python**	**C#**	**Java**	**F90**	**R**	**C++**
addition	+	+	+	+	+	+	+
substraction	-	-	-	-	-	-	-
multiplication	*	*	*	*	*	*	*
division	/	/	/	/	/	/	/
exponentiation	**	**	pow	pow	**	^	pow
increment	++	+=1	++	++		++	++
decrement	--	-=1	--	--		--	--
argument					Parameter::		
Parenthesis (to group expressions)	()	()	()	()	()	()	()
Equal to	==	==	==	==	.EQ.	==	==
Not equal to	!=	!=	!=	!=	.NE.	!=	!=
Less than	<	<	<	<	.LT.	<	<
Less or equal	<=	<=	<=	<=	.LE.	<=	<=
Greater than	>	>	>	>	.GT.	>	>
Greater or equal	>=	>=	>=	>=	.GE.	>=	>=
Logical NOT	*not*	*not*	!	!	.NOT.	!	!
Logical AND	*and*	*and*	&&	&&	.AND.	&&	&&
Logical OR	*or*	*or*	//	//	.OR.	||	||

Table S3

Precedence pecking order in CyML language and the languages currently supported by CyMLT.

Precedence	Language CyML	Python	C#	Java	F90	R	C++
	()	()	()	[] . ()	()	()	() [] -> .
	**	**	++ -- + (unary) - (unary) ! ~	++ -- + (unary) - (unary) ! ~	**	^ + -	! ++ -- + - * & (type) sizeof
	+= -=	+= -=	* / %	* / %	* /	%%	* / %
	* / // %	* / // %'	+ -	+ -	+ -	* /	+ -
	+ -	+ -	<< >>	<< >>	//	+ -	<< >>
	<< >>	<< >>	< <= > >=	< <= > >=	== /= < <= > >=	< > <= >= == !=	< <= > =>
	in not_in is is_not < <= > >= != ==	in not_in is is_not < <= > >= != ==	==, !=	== !=	.NOT.	!	== !=
	and	and	&&	&&	.AND.	&&	&&
	or	or	||	||	.OR.	||	||
			= += -= *= /= %= &= ^= |= <<= >>= >	= += -= *= /= %= &= ^= |= <<= >>=	.EQV. .NEQV.	=	= += -= *= /= %= &= ˆ= |= <<= >>=

Table S4

Mapping of built-in-functions between CyML and the languages supported by CyMLT.

Function	Language						
	CyML	**Python**	**C#**	**Java**	**F90**	**R**	**C++**
Exponential	exp(x)	exp(x)	Exp(x)	exp(x)	exp(x)	exp(x)	exp(x)
Natural log	log(x)	log(x)	Log(x)	log(x)	log(x)	log(x)	log(x)
Square root	sqrt(x)	sqrt(x)	Sqrt(x)	sqrt(x)	sqrt(x)	sqrt(x)	sqrt(x)
Power	pow(x,r)	pow(x,r)	Pow(x,r)	pow(x,r)	x**r	x^r	pow(x,r)
Absolute value	abs(x)	abs(x)	Abs(x)	abs(x)	abs(x)	abs(x)	fabs(x)
Smallest integer ($\geq$ x)	ceil(x)	ceil(x)	Ceiling(x)	ceil(x)	ceiling(x)	ceiling(x)	ceil(x)
Largest integer ($\leq$ x)	floor(x)	floor(x)	Floor(x)	floor	floor(x)	floor(x)	floor(x)
Division remainder	divmod()	divmod()	DivRem	floorMod	mod(x,y)	%%	fmod(x,y)
Rounding	round(x)	round(x)	Round(x)	round(x)	nint(x)	round(x)	round(x)
Cosinus	cos(x)	cos(x)	Cos(x)	cos(x)	cos(x)	cos(x)	Cos(x)
Sine	sin(x)	sin(x)	Sin(x)	sin(x)	sin(x)	sin(x)	sin(x)
Tangent	tan(x)	tan(x)	Tan(x)	tan(x)	tan(x)	tan(x)	tan(x)
Arc sine	asin(x)	asin(x)	Asin(x	asin(x)	asin(x)	asin(x)	asin(x)
Arc cosinus	acos(x)	acos(x)	Acos(x)	acos(x)	acos(x)	acos(x)	acos(x)
Arc tangente	atan(x)	atan(x)	Atan(x)	atan(x)	atan(x)	atan(x)	atan(x)

Table S5

Mapping of flow control statements between CyML and the languages supported by CyMLT.

Flow control statement	**Language**						
	CyML	**Python**	**C#**	**Java**	**F90**	**R**	**C++**
Conditionally execute statements	if	if	if { }	if { }	if end if	if { }	if { }
Loop a specific number of times	for k in range(n)	for k in range(n)	for(int k=0; k<n; k++) { }	for(int k=0; k<n; k++) { }	do k=0, n-1 end do	for(k in seq(0,n-1) { }	for k=0: n-1 { }
Loop an indefinite number of times	while	while	while { }	while { }	do while end do	while { }	while { }
Terminate and exit loop	break	break	break	break	exit	break	break
Skip a cycle in a loop	continue	continue	continue	continue	continue	next	cycle
Return to invoking function	return	return	return	return	return	return	return
Conditional alternate statements	else elif	else elif	else else if	else else if	else elseif	else else if	else else if
Conditional case selections	if elif	if elif	if { } else if { }	if { } else if { }	if { } elseif { }	if { }elseif { }	if { } else if { }

```
1  <ModelUnit modelid="SQ.WheatPhenology.ShootNumber" name="ShootNumber" timestep="1" version="1.0">
2      <Description>
3          <Title>CalculateShootNumber Model</Title>
4          <Authors>Pierre MARTRE</Authors>
5          <Reference>http://www1.clermont.inra.fr/siriusquality/?page_id=427</Reference>
6          <Abstract>calculate the shoot number and update the related variables if needed</Abstract>
7      </Description>
8      <Inputs>
9          <Input name="leafNumber" description="Leaf number " variablecategory="state" inputtype ="variable"
10             datatype="DOUBLE" min="0" max="10000" default="3.34" unit="leaf" />
11         ...
12     </Inputs>
13     <Outputs>
14         <Output name="canopyShootNumber" description="shoot number for the whole canopy"
15             variablecategory="state" datatype="DOUBLE" min="0" max="10000"  unit="shoot/m**2" />
16         ...
17     </Outputs>
18     <Function name = "fibonacci" language="Cyml" type="internal" filename="algo/pyx/Fibonacci.pyx"/>
19     <Algorithm  language="Cyml" platform="" filename="algo/pyx/ShootNumber.pyx" />
20     <Parametersets>
21         <Parameterset name="Pwheat1" description="some values in there" >
22             <Param name="targetFertileShoot">600.0</Param>
23             <Param name="sowingDensity">288.0</Param>
24         </Parameterset>
25      </Parametersets>
26      <Testsets>
27         <Testset name="check wheat model1" parameterset = "Pwheat1" description="some values in there" >
28             <Test name ="test_wheat1">
29                 <InputValue name="canopyShootNumber_t1" >288.0</InputValue>
30                 <OutputValue name="leafTillerNumberArray" >[1, 1, 1, 2]</OutputValue>
31                 ...
32             </Test>
33         </Testset>
34      </Testsets>
35 </ModelUnit>
```

Listing S1: A Crop2ML model specification for the shoot number model.

Listing S1 illustrates the main elements and format of Crop2ML model specification. All model inputs and outputs are not specified. Missing elements are replaced by "...". A parameter set called *Pwheat1* is defined and is used in the unit test *test_wheat1*. Other parameter sets can be specified and new unit tests can also be written for different parameter sets or for the same parameter sets. This model specification is parsed and a CyML code is generated (Supporting Information Listing S2). The inputs are the arguments of the generated function and the outputs are declared in the body of the function and are return variables. Finally, the algorithm part of the specification is incorporated in the body of the function.

Here, we present the *shootnumber* unit model, which is one model unit of the *WheatPhenology* Crop2ML composite model. The inputs of the model are as follows:

- canopyShootNumber_t1: Shoot number per square meter at the previous time step

- leafNumber: Number of appeared leaves present per main stem
- sowingDensity: Number of seeds per m²
- targetFertileShoot: Maximum number of shoots per m²
- NumberTillerCohort_t1: Number of tiller cohorts at the previous time step
- tilleringProfile_t1: List of the number of tillers per m² in each tiller cohort at the previous time step
- leafTillerNumberArray_t1: Array of the number of tillers in each leaf cohort.

The algorithm then updates the related variables such as *canopyShootNumber*, *tilleringProfile* and *leafTillerNumberArray* and estimates the number of tillers (*tillerNumber*) and the current number of shoots per plant (*averageShootNumberPerPlant*). The number of shoots per plant (*shoots*) is first calculated as a function of number of appeared leaves (*appeared Leaves*) on the main stem based on a Fibonacci series:

$$shoots_{emergedLeaves} = shoots_{emergedLeaves-1} + shoots_{emergedLeaves-2} \quad (1)$$

$$emergedLeaves \geq 3,\ shoots_1 = 1,\ shoots_2 = 1 \quad (2)$$

Then, the maximum number of shoots is limited by a threshold value (targetFertileShoot) by assuming that canopies have a constant maximal number of fertile shoots. So, the current shoot number at canopy level at a given time is given by:

$$canopyShootNumber = min\ (shoots * sowingDensity, targetFertileShoot) \quad (3)$$

A new tiller appears when the rate of the canopy shoot number (the difference between the current and the previous canopy shoot number) is different from 0 (Supporting Information Listing S2, line 26). Then, the number of appeared tiller cohorts per square is stored in a list (tilleringProfile; Supporting Information Listing S2, line 27).

The cohort of tillers for each leaf layer (*leafTillerNumberArray*) is then expanding with the one of previous time (Supporting Information Listing S2, line 31).

```
def model_shootnumber(float canopyShootNumber_t1=288.0,
                      float leafNumber=3.34,
                      float sowingDensity=288.0,
                      float targetFertileShoot=600.0,
                      list tilleringProfile_t1=[288.0],
                      list leafTillerNumberArray_t1=[1, 1, 1],
                      int numberTillerCohort_t1=1):
    """
     - Name: ShootNumber -Version: 1.0, -Time step: 1
     - Description:
                 * Title: CalculateShootNumber Model
                 * Reference: Modeling development phase in the
                   Wheat Simulation Model SiriusQuality...
     - inputs:
                 * name: canopyShootNumber_t1
                   ...
    """
    cdef float averageShootNumberPerPlant, canopyShootNumber
    cdef intlist leafTillerNumberArray, lNumberArray_rate
    cdef floatlist tilleringProfile
    cdef int numberTillerCohort, emergedLeaves, shoots, i
    emergedLeaves = max(1, ceil(leafNumber - 1.0))
    shoots = fibonacci(emergedLeaves)
    canopyShootNumber = min(shoots * sowingDensity, targetFertileShoot)
    averageShootNumberPerPlant = canopyShootNumber / sowingDensity
    if (canopyShootNumber != canopyShootNumber_t1):
        tilleringProfile = integr(tilleringProfile_t1,canopyShootNumber - canopyShootNumber_t1)
    numberTillerCohort = len(tilleringProfile)
    for i in range(len(leafTillerNumberArray_t1),ceil(leafNumber),1):
        lNumberArray_rate.append(numberTillerCohort)
    leafTillerNumberArray = integr(leafTillerNumberArray_t1, lNumberArray_rate)
    return  (averageShootNumberPerPlant, canopyShootNumber,
             leafTillerNumberArray, tilleringProfile, numberTillerCohort)

def fibonacci(int n):
    if n<=1: return n
    else: return fibonacci(n-1)+fibonacci(n-2)
```

Listing S2: CyML code of the shootnumber unit model of the WheatPhenology composite model.

Chapter 5. Crop modeling frameworks interoperability through bidirectional transformation

Abstract

The crop modeling community has always been concerned with the diversity and proliferation of modeling and simulation platforms. Many efforts have been provided to enhance collaborative model development tasks and to address exchange and reuse issues. However, the diversity of platforms breaks down the collaboration between different groups of crop modeling researchers. To address reuse issues, we have identified some concepts that made it possible to define a component specification language Crop2ML and a minimal domain language CyML for the description of associated algorithms regardless of platform specificities. A transformation system CyMLT has been defined to transform Crop2ML models into different languages and components compatible to platforms requirements. However, this system requires to transform first manually legacy components into Crop2ML. Moreover, no approach exists to maintain the consistency between the source and the target models. In this context, the objective of this work is to provide an automatic transformation system that transforms model components from PBM platforms in Crop2ML components and packages at a high level of abstraction (specification and algorithms) under certain constraints.

1. Introduction

The crop modeling community has always been concerned with the diversity and proliferation of modeling and simulation platforms (Holzworth *et al.*, 2014a). Many efforts have been provided to enhance collaborative model development tasks (Janssen et al., 2017) and to address exchange and reuse issues. However, the diversity of platforms breaks down the collaboration between different groups of crop modeling researchers (Donatelli and Rizzoli, 2008). In Chapter 3 we provided a model development approach to address model reuse. It is based on the definition of a shared crop modeling metalanguage used to describe the specifications of model and their composition, with the CyML language (Chapter 4) used to describe their corresponding algorithms. This representation of models has been developed to generate automatically various model components that conform to crop modelling frameworks through a transformation system. However, it first involves the manual transformation of legacy components into the Crop2ML framework. Inconsistencies between source and target models may occur when the models evolve over time (Mens, Van Straeten, et al., 2006). In the case where Crop2ML models are updated, the transformation system can generate new model components for each PBM platform it supports. However, a model builder can choose to update a component in its platform without changing the Crop2ML models or create a new component from its platform. It requires synchronizing the Crop2ML model and all platform's components. In this context, the objective of this work is to provide an automatic transformation system that transforms model components from PBM platforms in Crop2ML components and packages at a high level of abstraction (specification and algorithms) under certain constraints. This transformation system is the inverse of CyMLT (Chapter 5, Figure 1). This inverse transformation system goes beyond a source-to-source transformation since the target is not only a source code but also Crop2ML model specifications. Therefore, the output of CyMLTx is composed of the model specifications at a high level of abstraction, one or several associated algorithms (pseudo-code) in CyML which have fewer details compared to the model implementation (source code), and external functions. It, therefore, implies a reverse engineering step that transforms the low-level representation or concrete implementation to a high-level representation.

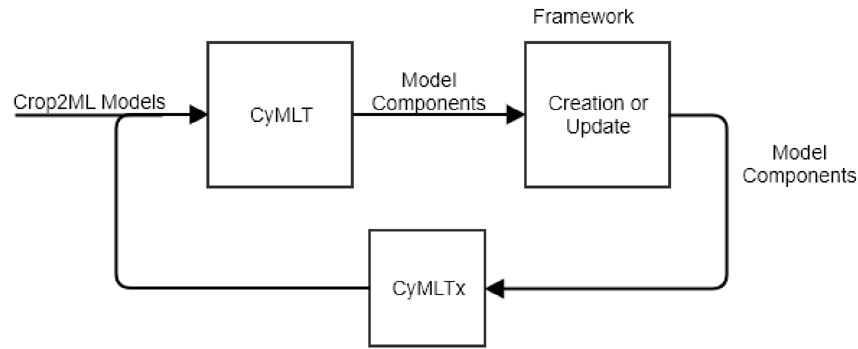

Figure 1: Crop2ML reengineering process.

Several research works invested in model transformation (Boukelkoul & Maamri, 2015; Ehrig & Ehrig, 2006; Kahani et al., 2019). Mens and Van Gorp (2006) proposed a taxonomy of model transformation with a notion of model that encompasses all levels of abstraction, including the source code as a model at a low level. Transformation approaches depend on the level of abstraction of the model (source code or specification) in the source or target. Reverse engineering (Mens, Van Gorp, et al., 2006) refers to transformations from a more concrete model to a more abstract model (e.g., from code to specification). Currently, there are no standard methods to achieve these transformations. Available methods are mainly based on compiler technology (such as parser generation), rather than modeling technology (Jiménez-Navajas et al., 2020; Kahani et al., 2019). Oda *et al.*, (2015) generated pseudo-code in English natural language from Python source code using Statistical Machine Translation (SMT) to improve program understanding. Although these approaches allow retrieving pseudo-code from the original code or generating its specification, they depend on the nature of the target model.

A source-to-source transformation, also named transcompiler, transpiler, or source-to-source compiler, is a process that converts source codes from a high-level language to another one. Although the first development of a transcompiler occurs during the 1950s-1960s, the full potential of this domain has not been realized yet. Bidirectional transformation can be used to keep two models synchronized and consistent (Biehl, 2010). To define a transpiler from Ada to Pascal and from Pascal to Ada, Albrecht *et al.*, (1980) used a subset of each language and converted them to the same intermediate representation, and thus provided a transformation definition for each direction. Other approaches used the same transformation definition for both directions (Yokoyama et al., 2008; Martins *et al.*, 2014). Martins *et al.* (2014) showed how inverted rewrite rules extended with additional rules and forward transformations could create, under certain circumstances, a total backward transformation. However, these approaches remain theoretical or applied for transformations between model specifications. More recently, AI-based approaches to address source-to-source transformation have been proposed. Lachaux *et al.* (2020) applied unsupervised machine

translation source code to create a transcompiler. The survey of Plaisted (2013) shows the great benefits of source-to-source transformation. It reveals that the use of an abstract language with a simple syntax and semantics facilitating its translation into many languages makes more effective the source-to-source transformation. Inspired by this observation, we hypothesize that CyML language that represents the intersection of several high-level languages can be leverage to enable bidirectional transformation between several programming languages.

In this work, we propose an approach based on reverse engineering (Chikofsky & Cross, 1990; Duffy & Malloy, 2005) and source-to-source transformation (Kulkarni, Chavan and Hardikar, 2015) that extend CyMLT. We called the extended part CyMLTx (CyMLT eXtended). It consists in transforming under some constraints a platform's model component to Crop2ML knowing that there are different platforms with different languages and software designs. It therefore requires providing a well-designed transformation mechanism that should be general enough to integrate any platform that satisfy our constraints. The system uses the Crop2ML specifications to bridge the difference between platforms. Crop2ML is independent of any given platform and its programming language. It is extensible, allowing new languages and frameworks to be added to the system without affecting it. Here, we first present the requirements of our system, its architecture and the different components such as the parser generator, the algorithm to infer the specification and the many-to-one transformation system. Then, we illustrate the transformation system using the BioMA framework as an example.

2. Methods

This section presents the design architecture of the transformation system and the main components that constitute the workflow of transformation. First, we enumerated some requirements and the properties of the system. Box 1 provides some definition of technical terms used in this section.

2.1. Requirements and properties

We designed CyMLTx to take into account the following properties:

- **Complete –** We defined CyML as an intersection of framework languages. Thus, the constructs of these languages are strictly limited to the common area defined by CyML. That is, any construct of any language has to be mapped to CyML constructs to avoid missing some transformations.

Box 1. Terminology and formal definitions.

A ***grammar*** is a set of rules or production rules that describes the syntax of a language. *Syntax* refers to a way a set of symbols may be combined to form a set of valid sentences (programs) in a language. *Semantics* reveals the meaning of the syntactical valid sentences.

A ***formal grammar*** of a language consists of a set of *terminal* symbols which is the alphabet defining the language, a set of *nonterminal* symbols which is used to generate the *words* in the language, a set of *production rules* (also called rules, rewriting rules or production) that describes how each nonterminal is defined in terms of terminal symbols and *nonterminal*, and a *specific nonterminal (start symbol)* that specifies the order of productions rules. Chomsky (1965) differentiated grammars according to the structure of their production rules. Context-free grammars allow linking a single nonterminal to a list of terminals and nonterminals in a production rule. Context free grammars can be used to formalize most of the rules describing syntactic structure.

Parsing or ***syntactic analysis*** refers to the process of finding the syntactic structure associated with an input sentence.

A ***Parser*** implements parsing algorithm. There are different kinds of parsers. The type of a parser depends on the type of the context free grammar that may be either a LR(k) or a LL(k) grammar. LR(k) grammars are those for which the LR(k) parsers read from left to right and look k input symbols beyond its current input position, producing thus a rightmost derivation. LL(k) grammars are those for which the LL(k) parsers read from left to right and look k input symbols to the right of its current input position, producing thus a leftmost derivation.

A ***parse tree*** *or* ***concrete syntax*** *tree* is the output of a *parser*.

An ***abstract syntax tree*** *(AST)* is a refinement or simplification of a parse tree, with some non-terminals, keywords, and punctuation removed while maintaining the meaning of the program.

An ***abstract semantic graph*** *(ASG)* is a refinement of AST with semantic information, namely type information.

A ***graph rewriting*** *rule* is a triple where the first two elements are graph patterns and the third element is a set of embedding descriptions which specifies how to substitute the second to the first when the rule is applied.

A ***transformation system*** is the automatic generation of a target model (may be a source code) from a source model (may be a source code), according to a *transformation definition*.

A ***transformation definition*** is a set of *transformation rules* that together describe how a model in the source language can be transformed into a model in the target language.

A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language. It can also be interpreted as a *rewriting rule*.

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

- Moreover, the specification of model components must allow inferring all information defined in Crop2ML, such as input, output type information, and authors.
- **Extensible -** The approach must be flexible to allow adding other languages or PBM platforms.
- **Efficient -** The complexity of the transformation algorithm must be as minimal as possible.
- **Platform-independent -** The implementation of the transformer can be deployed in various operating systems to be easily integrated as a plugin into other software projects.
- **Modular -** It is composed of independent modules used for specific tasks.

According to the model transformation taxonomy proposed by Mens and Van Gorp (2006), CyMLTx is:

- based on a source code (PBM platform model components) and an abstract model (Crop2ML);
- exogenous, that is transformations are between models expressed using different languages;
- is based on the properties of the source model;
- vertical, that is the source and target reside at different levels of abstraction.

2.2. Design architecture

Figure 2 shows the global architecture of the transformation system. It is based on ANTLR parser generator (Parr, 2013) that produces parsers for different grammars. Each parser of a specific language analyzes the model component written in this language and generates a concrete syntax tree (CST). The CST is transformed in abstract syntax trees (ASTs) that take into account the limited constructs of each language. All AST are then transformed into a unique representation of an abstract semantic graph (ASG). It means that a model implemented in different languages must have the same ASG. The architecture relies also on two other main components: Crop2ML extractor and source to source transformer. The Crop2ML extractor uses the ASTs to generate the Crop2ML model specifications under some constraints used to define patterns (Vasenin and Krivchikov, 2020), while the source-to-source transformer uses the ASG to generate the model algorithm (pseudo-code) based on a transformation workflow.

2.3. ANTLR parser generator and AST generator

A language parser is in charge of constructing the CST based on source components that it receives. For that, it is necessary to provide the grammar of each programming language that the system supports. It is cumbersome to implement a parser for each language for a scalable system in a context of multiple languages. Our approach uses the grammar of any language to generate its parser through an ANTLR Parser Generator. ANTLR is both a tool and a metalanguage allowing to express the LL(*) grammars of different languages (* means k is infinite). The ANTLR provides a collection of grammars for many popular programming languages including C, C++, Java, C#, Fortran, Python, and Ruby. It augments the grammars with tree operators, rewrite rules, and actions. The version 4 of the ANTLR generates a parser that produces the CST that contains all the information of the source code (Parr, 2013).

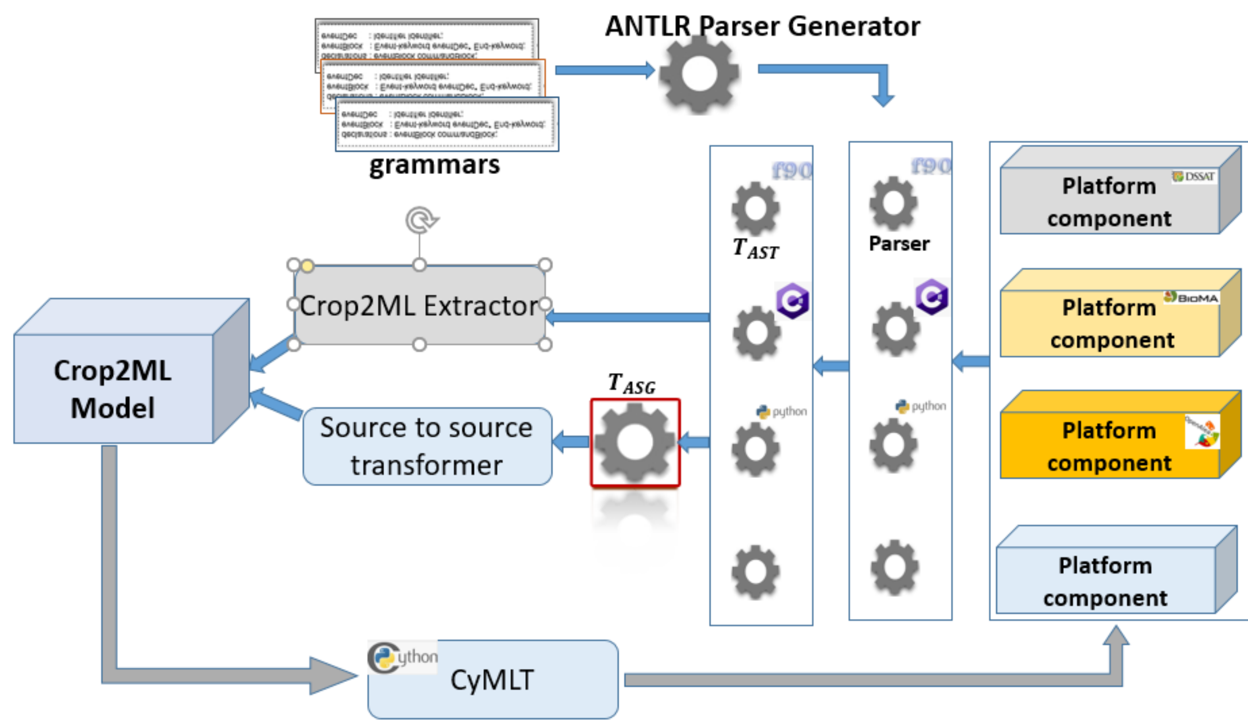

Figure 2: CyMLTx transformation system architecture. CyMLTx is based on the ANTLR parser generator and contains two functional subsystems: (i) a Crop2ML extractor that generates the model specification and (ii) a source-to-source transformer that generates model algorithms in CyML. The grey arrows show the CyMLT transformation process. The whole system leads to an interoperable system.

The process of AST generation consists in removing the information that is not necessary to maintain the structure of the code while preserving its meaning. The abstract syntax tree is therefore an isomorphism of the CST. It is at the level of this process that our approach guarantees the exclusive use of limited constructs. The use of a language construct that is not contained in the restriction generates an error message that indicate to the user that this construct is not allowed.

2.4. Extraction of model specifications

The Crop2ML extractor takes the ASTs generated from the component and builds Crop2ML model specifications based on the properties of the component. Our approach relies on the traversal of the AST nodes in order to detect recurring nodes that lead to the same structures and whose transformation allows retrieving the Crop2ML elements. To deal with the problem in a general way, we use an approach in three steps:

- define the formal specifications of the general transformation rule;
- identify the type of nodes whose transformation led to Crop2ML elements;
- transform these nodes and extract the Crop2ML elements.

2.4.1. Formal specifications of the general transformation rule

Let us call *primitives* a specification element in the source component. The general transformation rule consists in defining an extractor function that takes as inputs an internal AST node, searches and extracts the primitives to produce the model specifications. Thus, we can formally express the rule as follows as:

$$\boldsymbol{source} \xrightarrow{\boldsymbol{extractor}} \boldsymbol{target}$$

where,

- *source* is the internal node containing the *primitives;*
- *target* is the Crop2ML model specifications that corresponds to the source;
- *extractor* is the function used based on the primitives' constraints to generate the target.

This transformation rule is general and can be used for any primitive. However, it needs to be more specific and for that we proceed to the second step.

2.4.2. Identification of different types of nodes whose transformations lead to retrieve Crop2ML elements

Let us assume that there are recurring nodes that help to retrieve source primitives. We can therefore call *patterns* the relationships of these primitives with the corresponding Crop2ML elements. We define a formal rule of the pattern based on the primitives:

$$\boldsymbol{primitive} \xrightarrow{\boldsymbol{pattern}} \boldsymbol{element}$$

where,

- *primitive* is a terminal node which is a specification element in the source;
- *element* is an element of Crop2ML model specification that corresponds to the primitive;
- *pattern* evaluates if the primitive contains sufficient information to produce a Crop2ML model element.

A specific approach leads to identify the primitives in the AST. However, since our approach concerns various source components, we propose in Table 1 a list of generic names of primitives that will be identified in each source component after analyzing it.

Table 1

Identification of patterns used to produce Crop2ML elements.

Symbol	Pattern name	Identification
P1	Input sequence pattern	Identify a list of inputs with their attributes and values
P2	Output sequence pattern	Identify a list of outputs with their attributes and values
P3	Initialization pattern	Identify the initialization function
P4	Algorithm pattern	Identify the algorithm function
P5	External function sequence pattern	Identify the external functions used to express the algorithm
P6	Parameter sequence pattern	Identify unit tests and parameter values
P7	test sequence pattern	Identify unit tests and input output values

2.4.3. Generation of Crop2ML model specification

The identification of each pattern in the source component leads to apply a transformation on the primitives to produce a part of Crop2ML model. The concatenation of each result produces the entire model. However, the model specification provides the link where the initialization, external function and algorithm are expressed in CyML. The transformation of these primitives requires a source-to-source transformation system.

2.5. Many-to-one transformation system

CyMLT is the transformation system that exports a CyML code to other languages. Our intent is to design a system that automatically imports and exports model components from several crop modelling platforms. There are two possible ways to address this objective, either use the same transformation definition for both directions or define explicitly a transformation definition for each direction. We propose to analyze these propositions.

Let us consider a unidirectional transformation $M: A \rightarrow B$ (with A and B the domain and range of M, respectively) is a relation from A to B such that if $m(a) = b$ and $m(a) = c$ then $b = c$.

$M(a)$ is defined if there exists b in B such that $M(a) = b$

If $M(a)$is defined for all a in A, M is a total mapping. M is a partial mapping from A to B if it is only defined for a subset of B. We have designed CyML as an intersection of the languages of platforms such that CyMLT is a total mapping.

If $M: A \rightarrow B$ and for each b in B there is at most one a in A such that $M(a) = b$, then M is an injection (one-to-one transformation) from A into B. If M is a total mapping such that for each b in B there is exactly one a in A such that $M(a) = b$, then M is a bijection (one-to-one correspondence) between A and B. Such approach is in most situations much too restrictive (Poskitt et al., 2014) since it requires a total mapping that needs to restrict A and B.

If $M: A \rightarrow B$ is an injection, then we can find the inverse mapping $M^*: B \rightarrow A$ such that $M^*(b) = a$ if and only if $M(a) = b$. If there exists b in B for which there is no a in A such that $M(a) = b$, then M^* will be a partial function. Inconsistencies can arise between the target and source models since the inverse transformer omits some elements of the models. This asymmetry is observed when the target language is less restrictive and provides artifacts that cannot be mapped with the source language. It is therefore useful to restrict the target languages to the same domain as the one identified when defining CyML.

One flexible approach is the surjective mapping defined by:

If $M: A \rightarrow B$ and for each b in B there is at least one a in A such that $M(a) = b$, then M is surjective. Many constructs of the source language can be mapped with one construct of the target language. One example is the case where "while" and "for" statements could each be mapped with the "for" statement in the target language. In our case, the constructs of the source that satisfy this mapping are equivalent, hence the bijection mapping. However, in practice, only one construct in A is generated. We therefore propose a backward transformation distinct from the forward transformation.

Thus, the approach of the backward transformation is slightly different from the CyMLT one. Here, there are many sources to one target that we call many to one transformation. This means that all algorithm components implemented in different languages will be transformed into a model algorithm in CyML. CyMLTx follows these steps:

- Traverse the AST to identify the algorithm;
- Translate the sub AST into the CyML abstract semantic graph ASG;
- Generate CyML algorithm from the ASG.

As mentioned above, it is required to restrict the language to minimize inconsistencies. This restriction can be managed in two ways: either by restricting the grammar or by using the whole grammar and provide a mechanism which ensures that the restriction of language constructs is met. To propose a scalable system, it is better to consider the whole grammar and to provide a mechanism that can be progressively extended with new language constructs that satisfy the target language.

2.5.1. Identification of the Algorithm

The implementation of model components hides model algorithms that abstract the biophysical process. For example, a simple algorithm whose mathematical expression can be written in two lines can be implemented in a code of more than 30 lines since it embeds many PBM platform artifacts. However, platforms offer a fixed design that makes it possible to know exactly where the algorithm can be extracted. The information that enables this extraction is defined as a pattern that can be used to infer model algorithms. The initialization and external functions should also be identified and retrieved.

2.5.2. Transformation from abstract syntax tree to abstract semantic graph

Definition of the CyML Abstract Semantic Graph

The CyML ASG is a directed graph where the nodes are labelled based on the common constructs provided from the intersection of the framework languages.

Some constructs are assignment (e.g. for statement, if statement, while statement, or function statement). We propose a share and unique collections of statements (implemented as a node in the ASG) to have a unique ASG representation that will be a pivot for our system.

Let us build the ASG for an assignment statement $x = 5$:

ASG(f) = *Node* (type = “assignment”,

left = Node (type = “local”, name = “x”, pseudo_type = “int”, location = “1”),

right = Node (type = “int”, value = “5”, pseudo_type = “int”, location= “1”),

pseudo_type= “void”

location = “1”)

where *Node* is a prototype allowing to create the nodes and their links.

In this example, the graph contains three nodes (one parent node of type *assignment* and two children nodes) and two implicit edges (one from the root to the left node and another that links the root to the right node). The ASG guarantees some properties checked in the program. Each node of the ASG has a specific syntax based on the type of the node and a pseudo_type, which is a type provided by the type inference algorithm. This means that the approach is based on typing rules and consists in associating to each node the pseudo_type to allow the construction of types in a bottom-up way.

Transformation from abstract syntax tree to ASG abstract semantic graph

Here, we define a general formal approach that generates the ASG whatever the type of languages and PBM platforms.

The transformation from AST to ASG is a function based partially on a rewriting graph approach that transforms the AST generated from the CST to another graph (ASG) whose nodes are well defined. It consists in a set of rules formally represented as follows:

$M \rightarrow N$ where M and N are graphs.

These rules are applied on the AST if M is isomorph to a subgraph of the AST. M is isomorph to a subgraph of the AST if and only if this subgraph and M have the same type attribute. The AST is then rewritten recursively in a top-down way by replacing M's instance with N' s instance until M is a terminal node (i.e. a node without children).

The rewriting of the graph may be completed with another analysis on the symbol table (a data structure that stores information about source code identifiers) that depends on the philosophy of the source language like the scope of the variables. For example, in C#, a variable can be declared anywhere in the code but in the ASG all declarations are provided on the root. This requires to use the symbol table to add on each node of the ASG its declaration node, and if possible, the definition of new variables in order to preserve the semantics of the source code. These particularities led us to redefine the formal rule of the transformation as $M \xrightarrow{c} N$, where c is a constraint that must be satisfied to allow the transformation.

Using the same example, although we formalized a relationship for variable declaration, the transformation of the declaration node is possible if and only if it is on the root node, else the symbol table will be used. Thus, the constraint determines the applicability of the transformation rule.

2.5.3. From ASG to CyML Algorithm

Based on a visitor pattern using the type of each node of the ASG to select the corresponding visitor function, the CyML code is easily generated. We associated a portion of code to each node. The whole code is constructed by concatenating the portions of code associated with the descendant nodes. Thus, this transformation proceeds bottom-up (i.e. from the children toward the parent node).

3. Results: From BioMA to Crop2ML

The principles of the CyMLTx transformation system described above are general and support CyMLT implementation. The modularity of the architecture leads to implement a transformation workflow for each PBM platform. This workflow provides the following main modules:

- A parser module;
- A module that converts CST to AST;
- A module that rewrites AST to ASG;
- A module that generates the algorithm in CyML;
- A module that retrieves the model specification from the AST.

Here, we present some implementation elements with the Biophysical Models Applications (BioMA) platform, specifically on the SiriusQuality model development in BioMA. BioMA is a software framework designed and developed by the Joint Research Center (JRC) of the European Commission (Donatelli and Rizzoli, 2008). It is used for running, calibrating, and improving modeling solutions based on biophysical models. Models supported by the BioMA framework are implemented in C# using a component-based approach. BioMA is a flexible platform that allows model builders to add their artifacts in the model development. BioMA components development is based on strategy design pattern, which allows implementing alternative strategies for crop biophysical processes.

SiriusQuality is a process-based model that simulates the phenology and canopy development of small grain cereals and the fluxes of water, nitrogen and carbon in the soil-plant-atmosphere continuum in response to weather and crop management (Martre et al., 2006) . *SiriusQuality* components are implemented using the BioMA framework.

3.1. Abstract syntax tree generation

We implemented a module that transforms the CST from C# parser into an AST. First, the C# constructs that can be generated in CyML are listed and a visitor algorithm checks if the type of the CST node corresponds to one construct in the list. Otherwise, an error message is sent. For each node, a visitor function was implemented. It rewrites the node to be more readable and to ease the access to node elements. Listing 1 is an example of the transformation rule of additive expression. With the CST the access to the left and right members of the addition node requires to traverse the nodes. Our implementation gives a readable structure and facilitates their access.

```
def visit_additive_expression(self, node,multiplicative_expression,
                              PLUS,MINUS,location):
    if len(node.children)==1:
        return self.visit(multiplicative_expression)
    else:
        args = map(lambda n:self.visit(n), node.children)
        args = [arg[0] for arg in args]
        return reduceT(lambda x,y, op: {"type":"binary_op", "op":op,
                                        "left":x, "right":y,
                                       "pseudo_type":TYPED_API['operators'][op]
                                        (x['pseudo_type'],
                                         y['pseudo_type'])[-1]},
                       list(args))
```

Listing 1: Transformation of Addition node from CST to AST. This function eliminates the parenthesis and other addition node elements. It uses TYPED_API for type inference and generate a node with type, left, right and pseudo_type elements.

3.2. Abstract semantic graph and algorithm generation

The AST is transformed into ASG based on rewriting rules described for each node type. Each node is visited and the rule is applied until the terminal node is obtained. Listing 2 shows an example of rule on a binary operation node. The function *visit_binary_op* is the rewriting rule of a node of type *binary_op* (binary operation). Other rules are applied to all the children nodes of the *binary_op* node through a generic rule *visit* that calls the specific rule according to the type of the node.

```
def visit_binary_op(self, node):
    return {'type': 'binary_op',
      'op': str(node.op),
      'left': self.visit(node.left),
      'right': self.visit(node.right),
      'pseudo_type': self.visit(node.pseudo_type)}
```

Listing 2: Function that transform C# Assignment node to CyML ASG. A visit function is a generic rewriting rule that calls the specific visitor according to the type of node.

The binary operation visitor node can be generalized for ASG generation for several languages and platforms. However, the main differences are in binary operators, built-in methods and some particularities

of the platforms. Let us consider in the integration part of an example of vernalization process containing 704 lines of codes whereas the algorithm contains 47 (Listing 3).

```
private static void Integrate(States s1, States s, Rates r)
{
    s.VernalizationCum = s1.VernalizationCum + r.DailyVernalizationRate;
}
```

Listing 3: An example of integration function of vernalization strategy, s is an instance of states at current time step, s1 is an instance of state at current time.

BioMA uses domain classes to describe the domain of interest providing information about each variable used, and its value, a set of attributes such as minimum, maximum, default value, units, description. It provides a domain class for each interface variables declared as types with names such as States, Rates, Auxiliary, Exogenous etc. State class is instantiated for the previous and current time (or current and next time). The parsing of this integrate function (Listing 3) allows distinguishing the appropriate variable to adapt with CyML artifacts. That is, a variable of previous time is suffixed with "_t1" and the one of current by "_t" (Listing 4).

```
VernalizationCum_t = VernalizationCum_t1 + DailyVernalizationRate
```

Listing 4: Transformation of listing 3 in CyML algorithm.

3.3. Generation of Crop2ML model specification

The extraction of Crop2ML model specifications consists first in identifying the component files that contains specification elements. In SiriusQuality, each strategy class contains the descriptions of all parameters of the strategy, the algorithm identified with the method *CalculateModel*, and a list of inputs and outputs. The strategy class does not contain the description of the inputs and outputs. They are described in variable information domain classes files (*StatesVarInfo*, *RatesVarInfo*, …). In the design section, we identify some patterns that must be satisfied to allow specification retrieving. Let us analyze the inputs and outputs sequence patterns. These two patterns are built with both strategy classes files and variable information domain classes (VarInfo) files. Each pattern is a composition of two patterns. The first uses strategy class to identify the variables names. Locating in the constructor of the strategy class, a variable

name of a strategy is the value of the attribute "*PropertyName*" of one instance (prefixed by "*pd*") of the "*PropertyDescription*" class (Listing 5). This variable is an input if an instance prefixed by "_inputs" is previously declared, and it is an output if an instance prefixed by "_outputs" is previously declared. The first pattern is evaluated as true if all these constraints are identified. After retrieving these variable names, we identify the second pattern with VarInfo files, and the name of the variable. VarInfo class contains "*DescribeVariables*" method where the attributes (name, min, max, default value, …) of the variable are identified (Listing 6). This pattern is evaluated as true if all the attributes are identified. This analysis is continued for the other patterns.

```
List<PropertyDescription> _inputs = new List<PropertyDescription>();
PropertyDescription pd1 = new PropertyDescription();
pd1.DomainClassType = typeof(EC.JRC.MARS.Crop.CropML.Interfaces.Exogenous);
pd1.PropertyName = "AirTemperatureMaximum";
pd1.PropertyType = (( EC.JRC.MARS.Crop.CropML.Interfaces.ExogenousVarInfo.AirTemperatureMaximum))
pd1.PropertyVarInfo =( EC.JRC.MARS.Crop.CropML.Interfaces.ExogenousVarInfo.AirTemperatureMaximum)
```

Listing 5: Identification of the pattern to retrieve a variable name of a strategy

```
_AirTemperatureMaximum.Name = "AirTemperatureMaximum";
_AirTemperatureMaximum.Description = "Daily air temperature maximum";
_AirTemperatureMaximum.MaxValue = 55;
_AirTemperatureMaximum.MinValue = -30;
_AirTemperatureMaximum.DefaultValue = 25;
_AirTemperatureMaximum.Units = "°C";
_AirTemperatureMaximum.URL = "http://";
_AirTemperatureMaximum.ValueType = VarInfoValueTypes.GetInstanceForName("Double")
```

Listing 6: Identification of the pattern to retrieve the attributes of the variable identified with the first pattern.

```
<Inputs>
    <Input name="AirTemperatureMaximum" description="AirTemperatureMaximum"
        variablecategory="exogenous"  datatype="DOUBLE" default="25"
        min="-30" max="55" unit="degC" inputtype="variable" url="http://" />
...
</Inputs>
```

Listing 17: A portion of Crop2ML model specification result from a Component source code through inputs sequence pattern.

The validity of the generated model specification (a portion in Listing 7) and model algorithm is ensured with the Crop2ML DTD and the use of CyMLT to transform them and make unit tests.

4. Conclusion and perspectives

The goal of this study is to contribute to the exchange and reuse of process-based model components between crop modeling and simulation platforms. This paper completes the previous works that have defined an abstraction (Crop2ML) to allow representing a component regardless of platforms and designed a transformation system converting from Crop2ML to platforms. Here, we focused on the transformation from platforms model components to Crop2ML models. To achieve this objective, we designed a transformation system. This system is composed of the parser generator ANTLR4 and two subsystems. One subsystem retrieves the model specifications from the platforms under some constraints, and the second one generates model algorithms with limited constructs. The proposed approach has been partially developed for BioMA with minimum specifications to evaluate the architecture of the transformation system. However, this development can be extended to other platforms in order to build an interoperable system between platforms. The use of the ANTLR4 parser and the modularity of this system makes the system scalable.

The system proceeds to static analysis of model components by handling non compiled components (source code). This capacity to represent components as a graph and to traverse it enables to annotate model components and facilitates model curation and model element extraction for further processing.

In the area of reverse engineering, several metamodels are used to describe software for program comprehension, maintenance, and evolution. Washizaki et al., (2018) established a conceptual framework with definitions of different program metamodels and related concepts. In the future, it will be interesting to position our proposition of a platform of model components in relation to reverse engineering in this taxonomy.

This work is a further step towards the interoperability of PBM platforms by helping to facilitate the exchange of crop model components while maintaining the performance of the platforms. The whole Crop2ML framework has the intent to be a benchmark platform for evaluating the performance of the components of process-based crop models between crop modeling and simulation of platforms.

Chapter 6. General Discussion

6.1. Research findings

The main objective of this research is to *design a framework for PBM components exchange and reuse between modeling and simulation platforms. It provides a novel approach shared by PBM platforms to describe conceptual models (to address issue 1). It also enables model transformations to different languages and frameworks, and supports consistency between a conceptual model and its implementations (to address issue 2).* Based on our objective, we developed a global approach to provide an architecture of PBM component exchange and reuse (Crop2ML). Our approach aims to be integrative in order to arrive at a consensual solution and to take into account the differences between PBM platforms in terms of reuse.

6.1.1. A common modeling metalanguage for model specification

We addressed the lack of representation of conceptual models and the diverse interpretation of modularity in the frameworks. For that, we proposed a set of generic concepts that allow representing shared conceptual models between modeling frameworks. We assumed the white-box reuse approach goes through shared model concepts. These concepts should allow each modeler to derive the model implementation according to the specificities of its platform. This approach will change the habits of modelers that used model code as a knowledge base. The model concepts and algorithms should be understood by the whole modeling community to enhance crop modeling science. We proposed a metalanguage based on these shared concepts between crop simulation frameworks to describe specifications of model components and compositions. The composition relies on the graph-based modeling approach that facilitates model portability into any platform. Thus, we addressed model modularity at the conceptual level rather than at the implementation level. This declarative representation of model components increases its portability, allows model-based reasoning, and facilitates FAIR principles. The metalanguage contains unit tests concepts that will help modelers to integrate unit tests into model development. Our approach eases access to the scientific knowledge underlying the components. It links modeling and model documentation to avoid inconsistencies, knowing that the model specification is a sound basis for model documentation.

The systems biology community uses a similar approach to describe their models. This community provides several domain-specific modeling standard languages such as SBML, CellML, and NeuroML to exchange and store models (Cuellar et al., 2003; Gleeson et al., 2010; Hucka et al., 2003). These XML-

based languages provide specific elements to describe the model structure and equations using Mathematical Markup Language (MathML; Ausbrooks *et al.*, 2003). MathML language describes mathematical notations and captures both model structure and content. However, these languages are limited to specific formalisms (e.g., chemical reactions, differential equations). They cannot be easily extended to represent crop models in their full complexity and diversity.

The meaning of model variables and parameters needs to be fully understood before the model is reused or integrated into a large model component. This understanding is addressed with information on variables and parameters such as their name, unit, range of validity, description. Model documentation is often cited as a cure for understanding the data requirements of a model (Holzworth et *al.,* 2010). However, there is no common way to name model variables and parameters to facilitate composition with a model in other platforms. One approach is to provide a function that makes the correspondence of variables and parameters names between two models such that they coexist according to the data structure of the variable in the target platform.

6.1.2. CyML - A language for model algorithm

PBM components are usually described by finite difference equations and embed control structures such as loops or condition statements. Thus, there is not a clear mathematical formalism to describe them. Therefore, we have designed and implemented CyML to represent model component algorithms associated to model components specifications. 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 main interest of this language is to provide a common way to describe a process with the capacity to be integrated automatically in various 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 close to existing languages. The main source of complexity is in the model specifications. 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. CyML is a subset of Cython language. Its constructs come from the intersection of PBM platform's languages. CyML addresses several issues encountered in current PBM platforms, namely reproducibility, reusability, and transparency. It lowers the barrier of crop modeling platforms in terms of component reuse.

This choice of language limitation has its strengths and weaknesses. it reduces ambiguity in the language transformation since the base language (Cython) has some features that cannot be transformed into some target languages. CyML does not provide a formalism to link model components with data to build a modeling solution. Thus, the processes to read inputs, parameter values and write output values in a file is separated from the algorithm implementation given that it reduces reusability. The limitation of CyML may require adapting some model components provided by PBM platforms to support CyML constructs. It also reduces the capacity of the transformation system to infer a Crop2ML model from a platform's model component. Therefore, it poses the question of the completeness of the intersection of the platform's languages.

6.1.3. CYMLT: A transformation system between crop modeling platforms

Crop2ML concepts are at the heart of our framework, in particular the transformation system. We have designed two transformation definitions: from Crop2ML to PBM platforms (one-to-many) and from PBM platforms to Crop2ML (many-to-one), all based on a shared representation of abstract semantic graph (ASG) and graph rewriting rules. These two definitions led to an interoperability system between platforms. This approach of transformation based on Crop2ML reduces the complexity of the transformation algorithms. Let us consider n platforms. A direct side-by-side transformation system gives $A_n^2 = n(n-1)$ transformation definitions, while our transformation system provides $2n$ transformation definitions. As the number of platforms increases, the complexity of the direct transformation increases exponentially unlike in our approach. Moreover, our system has been designed to be extensible to other languages and platforms. The transformation from Crop2ML to PBM platforms forces modelers to provide documentations of their model components through a specification that will be used to produce model implementation. The transformation system from PBM platforms to Crop2ML forces platform developers to integrate into their structure of model components a pattern to identify model specifications. Automatic reuse is not possible without model specifications. It can be either specified in the code or as documentation. Model specification or documentation and source code must be more closely linked to infer the corresponding Crop2ML model. The transformation system enables users to focus on the scientific aspect of their model rather than on the internal knowledge of platform specificities. A model component can be reused, improved, integrated, and simulated in various platforms. It favors the diffusion of models, sharing them as software and scientific artifacts, and thus, enhancing transparency and reproducibility of crop models. The transformation system embeds platform specificities to generate automatically model components that conform to a specific

platform. It makes the complexity of component integration in different platforms identical with wide availability.

Although the transformation system allows establishing transformation definition with several languages (C#, Java, Python, C++, R, and Fortran) and platforms (DSSAT, BioMA, RECORD, SIMPLACE, OpenAlea), several limitations (complex data structure) required to extend it. These limits are related to the CyML language limitation and Crop2ML concepts. Moreover, it is easier to generate model components from Crop2ML than to infer a Crop2ML model from a platform model component. The reason is that a few modeling platforms (BioMA, SIMPLACE) provide a model specification that describes clearly model variables.

6.1.4. Facilitate model exchange and reuse through Crop2ML framework

The Crop2ML framework enables the exchange and reuse of PBM components between various PBM platforms through shared conceptual models. It provides a white-box reuse approach that could considerably increase the ability of modelers to share their model components. The main parts of this framework are:

- Crop2ML model package: a logical, standardized but flexible support to facilitate model sharing between modeling platforms through the definition of a directory structure;
- model specifications: a description of model components based on shared concepts between frameworks;
- model algorithm: a description of the behavior of model components in terms of sequence of inputs, successive rules or actions, conditions or a flow of instructions from inputs to outputs including mathematical expressions;
- model transformation: a transformation system allowing the import and export between Crop2ML and PBM platforms (DSSAT, BioMA, RECORD, SIMPLACE, OpenAlea, APSIM, and STICS);
- CropMStudio: A JupyterLab environment for Crop2ML to manage model life cycle such as creation, edition, transformation, composition, verification and validation (unit testing);
- Crop2ML Python library: an open, modular, and extensible library developed in Python that implements all the steps of the Crop2ML model lifecycle. It supports the current Crop2ML model specifications with the flexibility to be adapted for future versions. Other software projects can integrate it as a plug-in for adding its functionalities.

Some particularly novel features of Crop2ML framework include the parameterization of the grammar in each of the supported languages, modeling platform specific source code generation, the integration of

unit test into the model specification (allowing testing transformations), and platform/language specific documentation generation. Such features are likely to encourage verification, reproduction, and reuse among modelers and address the issue 1 framed in Chapter 1.

Other initiatives addressed model reuse by providing multi-scale and multi-language integrative frameworks such as OpenAlea (Pradal *et al.*, 2015), *Crops in silico* (Marshall-Colon et al., 2017) or OpenMI (Buahin & Horsburgh, 2018). These frameworks can compose and simulate heterogeneous models provided by different frameworks through a communication interface. The model components are often wrapped and represented as black-box components contrary to our approach. These frameworks enhance model reuse in their environment, but they do not address reusability between other simulation frameworks.

A shared model repository infrastructure is essential for efficient model exchange (Le Novere, 2006). Currently, model components are stored in a Github repository. In the future we need to provide a Crop2ML model repository to save models in a shared format to make them easily accessible and reusable by the PBM community. This repository aims at hosting alternative biophysical processes and facilitating model selection and improvement. It will help modelers to operate on multiple model components, compare them, or evaluate the impact of the integration of each component in large model components and crop modeling solutions.

6.1.5. Toward a standard in PBM component reuse

The Crop2ML framework, that is the model specification metalanguage, the model algorithm language and the 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 behavior has been demonstrated by Holzworth *et al.,* (2014b). However, despite the success of crop simulation platforms around which different communities are built, and some proposals of declarative language implementation (Athanasiadis & Villa, 2013; Rizzoli et al., 2008; Villa et al., 2006), the lack of a shared standard limits model reusability. This issue impedes the intercomparison and improvement of PBM. The availability of our framework through AMEI will allow building a large community around this system and can make Crop2ML a standard providing a means to compare independent biophysical processes or promote alternatives approaches.

However, we need to develop a strategy to benchmark PBM components between crop simulation platforms using Crop2ML. This strategy involves improving Crop2ML (e.g., data structure, metadata annotation, model selection, semantic composability), integrating other crop modeling and simulation

frameworks, making available the source codes of model components, and overcoming the issue of the intellectual property of model components.

6.2. Future research directions

This thesis presents some solutions to the problem of PBM component reuse between crop modeling and simulation platforms. A number of future directions for research were identified during the course of our work.

6.2.1. Improvements of Crop2ML framework

The limitations in our PBM exchange and reuse framework are twofold:

First, we have not provided any notion of composite variables or data structure. A composite variable is a variable made up of two or more variables or measures that are highly related to one another conceptually or statistically (Ley, 1972). The use of composite variables is a common practice in PBM development. A composite variable could be categorical variables (e.g. development stages) or cohorts of organs. In our current framework, we decompose first manually composite variables into several individual variables according to Crop2ML data structures. It leads to a semi-automatic transformation system. Thus, we need to focus on composite variables to target towards a complete automatic transformation system. Second, we do not provide any recommendation to select model components based on expert knowledge (Adam *et al.*, 2010b). Currently, model specification is the only source providing the modeling context through the provenance of the component and its description. We have no concept that allows ensuring the composition of contexts (R. Lara et al., 2006) or that guides the user to make a meaningful composition. Thus, it would useful to integrate into the Crop2ML framework a component selection approach based on expert knowledge to help building components compatible with the scientific requirements of crop modelling solutions.

The position of our framework with regard to systems engineering would require more investigations of previous works in Model-based development approaches. The SysML approach can be one of the future implementation area for the Crop2ML framework. It could help to formalize the different steps to exchange complex biophysical models.

6.2.2. Towards a multiscale framework?

A framework that combines PBM and FSPM

Crop2ML framework aims to enable the exchange and reuse of components between modeling platforms, notably between crop growth 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 (phytomer). The exchange (sharing) of model components between crop growth models 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 an 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, Crop2ML only allows representing processes as functions and does not consider the plant's structure. To extend Crop2ML to the FSPM community will require extending Crop2ML to support complex data structures such as 3D geometry and topology. The Multiscale Tree Graph (Godin & Caraglio, 1998) structure used in OpenAlea can be integrated in Crop2ML to represent the plant's topological structure. This would also facilitate the coupling of components operating at different scales (e.g. leaf area expansion at the organ level vs. photosynthesis at the canopy level) through a white-box approach.

A link between Crop2ML and integrative modeling platforms

The convergence of our approach of model reuse and reproducibility with other initiative, 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* initiative 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 PBM components across programming languages by running asynchronous models in parallel. It requires to write wrappers in the different languages to process the asynchronous messages to manage model inputs and outputs. Crop2ML 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.

6.2.3. Use of Crop2ML framework to compare and improve modeling solutions

The intercomparison and improvement of crop models requires proposing a generic model of computation for Crop2ML models. Despite the differences between crop modeling and simulation frameworks, we found some common features that enabled us to represent biophysical processes regardless of their specificities. We developed Crop2ML under the assumption that differences in the outputs of modeling solutions are due to the differences in the individual processes. However, the differences in models of computation (sequential model; [BioMA, SIMPLACE, DSSAT], data flow [OpenAlea], discrete event [Record]) could also have a strong impact on the simulation output of a model, but we did not consider it in this work. We need to support different models of computation into Crop2ML framework to achieve our objective. A complementary approach to our transformation system was demonstrated through the automated transformation of input files of four agricultural models (Samourkasidis et al., 2019) enabling the discovery and reuse of data across modelling solutions. Together with Crop2ML they could ensure that a complete model implementation and the associated data can be transformed between modelling solutions. They could also allow quantifying processes of crop models in a robust and repeatable manner.

6.2.4. Extend Crop2ML with semantic annotation

The capability to export from Crop2ML to PBM platforms is well performed since the transformation system is designed to support the specificities of the target platforms. However, the semantic of a Crop2ML model is about the shared concepts defined to describe at a high level a biophysical process designed as a discrete-time model. There is no semantic to support the description of each instance of Crop2ML concepts. For example, since there is no shared convention to name model variables or parameters, the integration of a component into a larger component of other platforms requires adapting the name of its variables. This would require annotating Crop2ML models to add semantic information in order to make a semantic link between any Crop2ML model variables or parameters, and those of components integrated in PBM platforms. This will also allow a semantic composability of Crop2ML models instead of a syntactic composability that analyzes if the pair of variables to be linked are isomorphic. Our perspective is to provide a metadata annotation based on a minimum requirement to annotate each element of Crop2ML models with

relevant information to avoid component misuse, and to allow the distribution of Crop2ML models via a shared repository, like BioModels (Glont et al., 2018; Le Novere, 2006). This annotation should use and extend existing ontologies such as the *Crop Ontology* (Matteis et al., 2013), or the *Agronomy ontology* (Jonquet et al., 2018). The annotation of model variables and parameters would also greatly facilitate the link between crop model components and modeling solutions and data.

The annotated model components can then be interoperable at the semantic level offering the capability to automate their composition through Semantic Web technologies and web services. The semantic interoperability ensures that the composition is biologically meaningful and consistent with the modeling objective defined by the modelers. To address this issue, it would be interesting to explore Automatic web service composition research area (Bekkouche et al., 2017; Hatzi et al., 2012; Netedu et al., 2020). This area promotes the improvement of the composition of multiple Web services to create new ones with specific functionality. It requires efficient automated service discovery and selection approaches (Geem et *al.*, 2001; Azmeh *et al.*, 2011) from a web-based repository of Crop2ML model components followed by the validation of the semantic composability (Mahmood et al., 2012; Szabo & Teo, 2009) that involves information about the components' behavior.

Conclusion

In this thesis we study the issue of the reuse of process-based model (PBM). PBM are increasingly used to analyze and predict the response of agricultural systems to climatic, agronomic and, more recently genetic, factors. They have often been developed in crop simulation platforms to ensure their future extension and to couple different crop models with a soil model and a crop management event scheduler. The emergence of crop modeling frameworks has considerably increased the use of crop models in research, as well as their applications for the management of production systems or scenario analysis. Despite their advances, these frameworks have negatively impacted models by causing a loss of transparency for modelers. As stated in Chapter 2 the main drawback of the wide range of frameworks is their difference in terms of programming languages, software designs, and architecture constraints. This led to a decrease in the development of new formalisms, particularly for new uses related to phenotyping. Indeed, the dependency on the frameworks limits model reuse between them. It also adds the divergent consideration of modularity, the lack of the description of models, which do not promote their reuse. Even if model reuse

has already been studied, few improvements have been made (Holzworth *et al.*, 2014a) with little collaborative effort.

This research proposes a comprehensive framework for process-based model components exchange and reuse. It provides an approach that automatically generates model components with flexible structures using shared modeling concepts. By “flexible structures”, we mean that model components are not generated by a specified model structure, but the model component has a structure that is dynamically constructed according to the target framework.

I proposed a shared representation of model components to address reuse. Using this shared representation has three advantages: (1) it provides a good understanding of the model component by different parties (2) it serves as a bridge between modeling frameworks (3) it facilitates the extension of the framework with other parties. Several frameworks have reused the same components implemented in Crop2ML using the CyMLT transformation system. A shared representation reduces the number of transformations between frameworks and avoids providing a specific transformation between each pair of frameworks. This could reduce the genericity of our approach and increases the complexity of the transformation. The case study has been illustrated with international platforms which are highly different: DSSAT, BioMA, SIMPLACE, RECORD, and OpenAlea. We have also shown the extensibility of our framework with a widely used platform APSIM.

Our framework differs from the existing crop modeling frameworks. It highlights the strong need to go through a conceptual approach from which the implementation must derive, while keeping consistency between them. This entails that there is an information gap between a model component as such and how it is implemented in terms of language, structure, and software design. Our approach does not depend on any implementation language and satisfies the requirements of existing frameworks. Thus, the aim of our framework is not to replace existing crop modeling platforms but to exchange components between them while preserving the constraints of existing platforms (e.g. programming languages, programming paradigm).

From the initial research questions and the final research results, we conclude that this research addresses the identified issues in crop model reuse. We tested the applicability of the proposed framework by developing a prototype. The proposed framework bridges the gap between modeling and simulation platforms in model component reuse.

Potential users of these results are all the actors in software development and in the crop simulation model development, including modelers, model builders, crop modeling framework developers. It aims at improving the conceptual modeling stage and increasing the reuse of crop model components. The focus on

the conceptual modeling stage in crop modeling and the definition of transformation systems that make consistent the conceptual model and its implementations grant the originality of this research, because this subject has not been adequately studied before in the crop modeling and simulation community for reuse purposes.

References

Adam, M., Belhouchette, H., Corbeels, M., Ewert, F., Perrin, A., Casellas, E., Celette, F., & Wery, J. (2012a). Protocol to support model selection and evaluation in a modular crop modelling framework: An application for simulating crop response to nitrogen supply. *Computers and Electronics in Agriculture*, *86*, 43–54. https://doi.org/10.1016/j.compag.2011.09.009

Adam, M., Corbeels, M., Leffelaar, P. A., Van Keulen, H., Wery, J., & Ewert, F. (2012b). Building crop models within different crop modelling frameworks. *Agricultural Systems*, *113*, 57–63. https://doi.org/10.1016/j.agsy.2012.07.010

Adam, M., Ewert, F., Leffelaar, P. A., Corbeels, M., van Keulen, H., & Wery, J. (2010a). CROSPAL, software that uses agronomic expert knowledge to assist modules selection for crop growth simulation. *Environmental Modelling and Software*, *25*(8), 946–955. https://doi.org/10.1016/j.envsoft.2010.02.007

Adam, Myriam. (2010b). *A framework to introduce flexibility in crop modelling: from conceptual modelling to software engineering and back*. PhD thesis. Plant Production Systems. Wageningen University, Wageningen, pp. 190.

Addiscott, T. M., & Whitmore, A. P. (1991). Simulation of solute leaching in soils of differing permeabilities. *Soil Use and Management*, *7*(2), 94–102. https://doi.org/10.1111/j.1475-2743.1991.tb00856.x

Akeret, J., Gamper, L., Amara, A., & Refregier, A. (2015). HOPE: A Python just-in-time compiler for astrophysical computations. *Astronomy and Computing*, *10*, 1–8. https://doi.org/10.1016/j.ascom.2014.12.001

Akin, E. (2003). *Object-Oriented Programming via Fortran 90/95,* 1317(1), Cambridge University Press, Cambridge, 348pp. https://doi.org/10.1017/CBO9780511530111

Albrecht, P. F., Garrison, P. E., Graham, S. L., Hyerle, R. H., Ip, P., & Krieg-Brückner, B. (1980). Source-to-source translation: ADA to Pascal and Pascal to ADA. *Proceedings of the ACM-SIGPLAN Symposium on Ada Programming Language, SIGPLAN 1980*, 183–193. https://doi.org/10.1145/948632.948658

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A pattern language*: *Towns, Buildings, Construction*, Oxford University Press, New York

Andrei, O., & Kirchner, H. (2009). A Port Graph Calculus for Autonomic Computing and Invariant Verification. *TERMGRAPH 2009, 5th International Workshop on Computing with Terms and Graphs, Satellite Event of ETAPS 2009*. https://hal.inria.fr/inria-00418560

Antle, J. M., Basso, B., Conant, R. T., Godfray, H. C. J., Jones, J. W., Herrero, M., Howitt, R. E., Keating, B. A., Munoz-Carpena, R., Rosenzweig, C., Tittonell, P., & Wheeler, T. R. (2017). Towards a new generation of agricultural system data, models and knowledge products: Design and improvement. *Agricultural Systems*, *155*, 255–268. https://doi.org/10.1016/j.agsy.2016.10.002

Argent, R. M., Voinov, A., Maxwell, T., Cuddy, S. M., Rahman, J. M., Seaton, S., Vertessy, R. A., & Braddock, R. D. (2006). Comparing modelling frameworks - A workshop approach. *Environmental Modelling and Software*, *21*(7), 895–910. https://doi.org/10.1016/j.envsoft.2005.05.004

Aslam, M. A., Ahmed, M., Stöckle, C. O., Higgins, S. S., ul Hassan, F., & Hayat, R. (2017). Can growing

degree days and photoperiod predict spring wheat phenology? *Frontiers in Environmental Science*, *5*(SEP), 1–10. https://doi.org/10.3389/fenvs.2017.00057

Asseng, S., Ewert, F., Rosenzweig, C., Jones, J. W., Hatfield, J. L., Ruane, A. C., Boote, K. J., Thorburn, P. J., Rötter, R. P., Cammarano, D., Brisson, N., Basso, B., Martre, P., Aggarwal, P. K., Angulo, C., Bertuzzi, P., Biernath, C., Challinor, A. J., Doltra, J., … Wolf, J. (2013). Uncertainty in simulating wheat yields under climate change. *Nature Climate Change*, *3*(9), 827–832. https://doi.org/10.1038/nclimate1916

Athanasiadis, I. N., Rizzoli, A. E., Donatelli, M., & Carlini, L. (2011). Enriching environmental software model interfaces through ontology-based tools. *International Journal of Applied Systemic Studies*, *4*(1–2), 94–105. https://doi.org/10.1504/IJASS.2011.042205

Athanasiadis, I. N., & 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*, 27–32. https://doi.org/10.1145/2541928.2541934

Ausbrooks, R., Buswell, S., Carlisle, D., Dalmas, S., Devitt, S., Diaz, A., Frou-mentin, M., Hunter, R., Ion, P., Kohlhase, M., Miner, R., Poppelier, N., Smith, B., Soiffer, N., Sutor, R., & Watt, S. (2003). *Mathematical Markup Language (MathML) Version 2 . 0 (Second Edition)*. In World WideWeb Consortium recommendation. http://www.w3.org/TR/MathML2/

Azmeh, Z., Driss, M., Hamoui, F., Huchard, M., Moha, N., & Tibermacine, C. (2011). Selection of composable web services driven by user requirements. *Proceedings - 2011 IEEE 9th International Conference on Web Services, ICWS 2011*, 395–402. https://doi.org/10.1109/ICWS.2011.47

Banks, C. M. (2010). Introduction to Modeling and Simulation. In *Modeling and Simulation Fundamentals* (pp. 1–24). John Wiley & Sons, Inc. https://doi.org/10.1002/9780470590621.ch1

Barbier, G., Cucchi, V., & Hill, D. R. C. (2015). Model-driven engineering applied to crop modeling. *Ecological Informatics*, *26*(P2), 173–181. https://doi.org/10.1016/j.ecoinf.2014.05.004

Basso, B., Cammarano, D., & Carfagna, E. (2013). Review of Crop Yield Forecasting Methods and Early Warning Systems. *The First Meeting of the Scientific Advisory Committee of the Global Strategy to Improve Agricultural and Rural Statistics*, 1–56. https://doi.org/10.1017/CBO9781107415324.004

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

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2011). Cython: The Best of Both Worlds. *Computing in Science & Engineering*, *13*(2), 31–39. https://doi.org/10.1109/MCSE.2010.118

Bekkouche, A., Mohamed Benslimane, S., Huchard, M., Tibermacine, C., Hadjila, F., Merzoug, M., & Mohammed BENSLIMANE, S. (2017). QoS-aware optimal and automated semantic web service composition with user's constraints. *Constraints. Service Oriented Computing and Applications*, *11*(2), 183–201. https://doi.org/10.1007/s11761-017-0205-1ï

Bergez, J. E., Chabrier, P., Gary, C., Jeuffroy, M. H., Makowski, D., Quesnel, G., Ramat, E., Raynal, H., Rousse, N., Wallach, D., Debaeke, P., Durand, P., Duru, M., Dury, J., Faverdin, P., Gascuel-Odoux, C., & Garcia, F. (2013). An open platform to build, evaluate and simulate integrated models of farming and agro-ecosystems. *Environmental Modelling and Software*, *39*, 39–49. https://doi.org/10.1016/j.envsoft.2012.03.011

Bergez, Jacques Eric, Raynal, H., Joannon, A., Casellas, E., Chabrier, P., Justes, E., Quesnel, G., &

Véricel, G. (2016). A new plug-in under RECORD to link biophysical and decision models for crop management. *Agronomy for Sustainable Development*, *36*(1), 1–8. https://doi.org/10.1007/s13593-016-0357-y

Biehl, M. (2010). Literature Study on Model Transformations. In: *Royal Institute of Technology*, 1–24.

Biggerstaff, T. J. (1989). *Software reusability: vol. 1, concepts and models* (A. J. Perlis (ed.)). ACM. https://doi.org/10.1145/73103

Bindi, M., Palosuo, T., Trnka, M., & Semenov, M. (2015). Modelling climate change impacts on crop production for food security. *Climate Research*, *65*, 3–5. https://doi.org/10.3354/cr01342

Boote, K., Jones, J. W., & Pickering, N. (1996). Potential Uses and Limitations of Crop Models (AJ). *Agronomy Journal*, *88*, 704–716. https://doi.org/10.2134/agronj1996.00021962008800050005x

Boudon, F., Pradal, C., Cokelaer, T., Prusinkiewicz, P., & Godin, C. (2012). L-Py: An L-System Simulation Framework for Modeling Plant Architecture Development Based on a Dynamic Language. *Frontiers in Plant Science*, *3*, 1–20. https://doi.org/10.3389/fpls.2012.00076

Boukelkoul, S., & Maamri, R. (2015). *Optimal Model Transformation of BPMN to DEVS*. In: Proceedings of the Conference on Computer Systems and Applications, AICCSA-12, Marrakesh; 1-8.

Brisson, N., Launay, M., Mary, B., & Beaudoin, N. (2010). Conceptual Basis, Formalisations and Parameterization of the Stics Crop Model. *Editons Quae*, Versailles. Collection “Update Science and Technologies”, 297 pp. http://www.quae.com/en/r1291-conceptual-basis-formalisations-and-parameterization-of-the-stics-crop-model.html

Brisson, N., Mary, B., Ripoche, D., Jeuffroy, M. H., Ruget, F., Nicoullaud, B., Gate, P., Devienne-Barret, F., Antonioletti, R., Durr, C., Richard, G., Beaudoin, N., Recous, S., Tayot, X., Plenet, D., Cellier, P., Machet, J.-M., Meynard, J. M., & Delécolle, R. (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), 311–346. https://doi.org/10.1051/agro:19980501

Brown, A. W. (2000). Large-Scale, Component-Based Development. Prentice Hall.

Brown, A. W. (2004). Model driven architecture: Principles and practice. *Software and Systems Modeling*, 314–327. https://doi.org/10.1007/s10270-004-0061-2

Brown, H. E., Huth, N. I., Holzworth, D. P., Teixeira, E. I., Zyskowski, R. F., Hargreaves, J. N. G., & Moot, D. J. (2014). Plant Modelling Framework: Software for building and running crop models on the APSIM platform. *Environmental Modelling and Software*, *62*, 385–398. https://doi.org/10.1016/j.envsoft.2014.09.005

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

Buahin, C. A., & Horsburgh, J. S. (2018). Advancing the Open Modeling Interface (OpenMI) for integrated water resources modeling. *Environmental Modelling and Software*, *108*, 133–153. https://doi.org/10.1016/j.envsoft.2018.07.015

Bucchiarone, A., Cabot, J., Paige, R. F., & Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, *19*(1), 5–13. https://doi.org/10.1007/s10270-019-00773-6

Bunniss, S., & Kelly, D. R. (2010). Research paradigms in medical education research. *Medical Education*, *44*(4), 358–366. https://doi.org/10.1111/j.1365-2923.2009.03611.x

Bysiek, M., Drozd, A., & 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*, 9–18. https://doi.org/10.1109/PyHPC.2016.006

Cary, J. R., Shasharina, S. G., Cummings, J. C., Reynders, J. V. W., & Hinker, P. J. (1997). Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, *105*(1), 20–36. https://doi.org/10.1016/S0010-4655(97)00043-X

Challinor, A. J., Wheeler, T. R., Craufurd, P. Q., Slingo, J. M., & Grimes, D. I. F. (2004). Design and optimisation of a large-area process-based model for annual crops. *Agricultural and Forest Meteorology*, *124*(1–2), 99–120. https://doi.org/10.1016/j.agrformet.2004.01.002

Chenu, K., Porter, J. R., Martre, P., Basso, B., Chapman, S. C., Ewert, F., Bindi, M., & Asseng, S. (2017). Contribution of Crop Models to Adaptation in Wheat. *Trends in Plant Science*, *22*(6), 472–490. https://doi.org/10.1016/j.tplants.2017.02.003

Chikofsky, E. J., & Cross, J. H. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software*, *7*(1), 13–17. https://doi.org/10.1109/52.43044

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. Cambridge: MIT Press.

Clark, A. M. (1998). The qualitative-quantitative debate: Moving from positivism and confrontation to post-positivism and reconciliation. *Journal of Advanced Nursing*, *27*(6), 1242–1249. https://doi.org/10.1046/j.1365-2648.1998.00651.x

Cohen-Boulakia, S., Belhajjame, K., Collin, O., Chopard, J., Froidevaux, C., Gaignard, A., Hinsen, K., Larmande, P., Bras, Y. Le, Lemoine, F., Mareuil, F., Ménager, H., Pradal, C., & Blanchet, C. (2017). Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Computer Systems*, *75*, 284–298. https://doi.org/10.1016/j.future.2017.01.012

Corbeels, M., McMurtrie, R. E., Pepper, D. A., & O'Connell, A. M. (2005). A process-based model of nitrogen cycling in forest plantations: Part I. Structure, calibration and analysis of the decomposition model. *Ecological Modelling*, *187*(4), 426–448. https://doi.org/10.1016/j.ecolmodel.2004.09.005

Cuadrado, J. S., & Molina, J. G. (2007). Building Domain-Specific Languages for Model-Driven Development. *IEEE Software*, *24*(5), 48–55. https://doi.org/10.1109/MS.2007.135

Cuellar, A. A., Lloyd, C. M., Nielsen, P. F., Bullivant, D. P., Nickerson, D. P., & Hunter, P. J. (2003). An Overview of CellML 1.1, a Biological Model Description Language. *SIMULATION*, *79*(12), 740–747. https://doi.org/10.1177/0037549703040939

de Lara, J., & Vangheluwe, H. (2004). Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, *15*(3–4), 309–330. https://doi.org/10.1016/j.jvlc.2004.01.005

de Wit, A., Boogaard, H., Fumagalli, D., Janssen, S., Knapen, R., van Kraalingen, D., Supit, I., van der Wijngaart, R., & van Diepen, K. (2019). 25 years of the WOFOST cropping systems model. *Agricultural Systems*, *168*(October 2017), 154–167. https://doi.org/10.1016/j.agsy.2018.06.018

de Wit, C. T. (1965). Photosynthesis of leaf canopies. In *Agricultural Research Reports* (Issue 663, pp. 1–54). https://doi.org/10.2172/4289474

Degueule, T., Combemale, B., Blouin, A., Barais, O., & Jézéquel, J.-M. (2015). Melange: A Meta-

language for Modular and Reusable Development of DSLs. *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, 25–36. https://doi.org/10.1145/2814251.2814252

Détienne, F. (1991). Reasoning from a schema and from an analog in software code reuse. In J. Koenemann-Belliveau, T. Moher & S.P. Robertson (Eds), *Empirical Studies of Programmers: Fourth WorkshopNew, Brunswick, NJ, USA,* 16.

Dhami, H. P. S. (2012). Composability of components in Component Based Software Development (CBD). *International Journal of Enterprise Computing and Business Systems*, *2*(1), 1–17.

Donatelli, M., Bregaglio, S., Confalonieri, R., De Mascellis, R., & Acutis, M. (2014). A generic framework for evaluating hybrid models by reuse and composition - A case study on soil temperature simulation. *Environmental Modelling and Software*, *62*, 478–486. https://doi.org/10.1016/j.envsoft.2014.04.011

Donatelli, M., Cerrani, I., Fanchini, D., Fumagalli, D., & Rizzoli, A. E. (2012). Enhancing model reuse via component-centered modeling frameworks: The vision and example realizations. *IEMSs 2012 - Managing Resources of a Limited Planet: Proceedings of the 6th Biennial Meeting of the International Environmental Modelling and Software Society*, 1185–1192.

Donatelli, M., & Rizzoli, A. E. (2008). A design for framework-independent model components of biophysical systems. *4th Biennial Meeting of International Congress on Environmental Modelling and Software: Integrating Sciences and Information Technology for Environmental Assessment and Decision Making, IEMSs 2008*, *2*, 727–734. http://www.scopus.com/inward/record.url?eid=2-s2.0-70349563625&partnerID=40&md5=251da64e55d9bed564ab136bf25897d7

Donatelli, M., Russell, G., Rizzoli, A. E., Acutis, M., Adam, M., Athanasiadis, I. N., Balderacchi, M., Bechini, L., Belhouchette, H., Bellocchi, G., Bergez, J.-E., Botta, M., Braudeau, E., Bregaglio, S., Carlini, L., Casellas, E., Celette, F., Ceotto, E., Charron-Moirez, M. H., … Zerourou, A. (2010). A Component-Based Framework for Simulating Agricultural Production and Externalities. In *Environmental and Agricultural Modeling:* (pp. 63–108). Springer Netherlands. https://doi.org/10.1007/978-90-481-3619-3_4

Dourado-Neto, D., Teruel, D. a., Reichardt, K., Nielsen, D. R., Frizzone, J. a., & Bacchi, O. O. S. (1998). Principles of crop modeling and simulation: I. uses of mathematical models in agricultural science. *Scientia Agricola*, *55*(spe), 46–50. https://doi.org/10.1590/S0103-90161998000500008

Duffy, E. B., & Malloy, B. A. (2005). A language and platform-independent approach for reverse engineering. *Proceedings - Third ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2005*, *2005*, 415–422. https://doi.org/10.1109/SERA.2005.9

Ehrig, H., & Ehrig, K. (2006). Overview of formal concepts for model transformations based on typed attributed graph transformation. *Electronic Notes in Theoretical Computer Science*, *152*(1–2), 3–22. https://doi.org/10.1016/j.entcs.2006.01.011

Enders, A., Diekkrüger, B., Laudien, R., Gaiser, T., & Bareth, G. (2010). The IMPETUS Spatial Decision Support Systems. In *Impacts of Global Change on the Hydrological Cycle in West and Northwest Africa* (pp. 360–393). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-12957-5_11

Fernique, P., & Pradal, C. (2018). Auto WIG: Automatic generation of python bindings for C++ libraries. *PeerJ Computer Science*, *2018*(4), e149. https://doi.org/10.7717/peerj-cs.149

Fournier, C., Pradal, C., Louarn, G., Combes, D., Soulié, J.-C., Luquet, D., Boudon, F., & Chelle, M. (2010). *Building modular FSPM under OpenAlea: concepts and applications*. https://hal.inria.fr/hal-

01192232v2

Gaiser, T., Perkons, U., Küpper, P. M., Kautz, T., Uteau-Puschmann, D., Ewert, F., Enders, A., & Krauss, G. (2013). Modeling biopore effects on root growth and biomass production on soils with pronounced sub-soil clay accumulation. *Ecological Modelling*, *256*, 6–15. https://doi.org/10.1016/j.ecolmodel.2013.02.016

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*. https://doi.org/10.1016/b978-012663315-3/50005-8

Gaudio, N., Escobar-Gutiérrez, A. J., Casadebaig, P., Evers, J. B., Gérard, F., Louarn, G., Colbach, N., Munz, S., Launay, M., Marrou, H., Barillot, R., Hinsinger, P., Bergez, J.-E., Combes, D., Durand, J.-L., Frak, E., Pagès, L., Pradal, C., Saint-Jean, S., … Justes, E. (2019). Current knowledge and future research opportunities for modeling annual crop mixtures. A review. *Agronomy for Sustainable Development*, *39*(2), 20. https://doi.org/10.1007/s13593-019-0562-6

Geem, Z. W., Kim, J., & Loganathan, G. V. (2001). A New Heuristic Optimization Algorithm: Harmony Search. *Simulation*, *76*, 60–68.

Giller, K. E., Tittonell, P., Rufino, M. C., van Wijk, M. T., Zingore, S., Mapfumo, P., Adjei-Nsiah, S., Herrero, M., Chikowo, R., Corbeels, M., Rowe, E. C., Baijukya, F., Mwijage, A., Smith, J., Yeboah, E., van der Burg, W. J., Sanogo, O. M., Misiko, M., de Ridder, N., … Vanlauwe, B. (2011). Communicating complexity: Integrated assessment of trade-offs concerning soil fertility management within African farming systems to support innovation and development. *Agricultural Systems*, *104*(2), 191–203. https://doi.org/10.1016/j.agsy.2010.07.002

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., Morse, T. M., Davison, A. P., Ray, S., Bhalla, U. S., Barnes, S. R., Dimitrova, Y. D., & Silver, R. A. (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), 1–19. https://doi.org/10.1371/journal.pcbi.1000815

Glont, M., Nguyen, T. V. N., Graesslin, M., Hälke, R., Ali, R., Schramm, J., Wimalaratne, S. M., Kothamachu, V. B., Rodriguez, N., Swat, M. J., Eils, J., Eils, R., Laibe, C., Malik-Sheriff, R. S., Chelliah, V., Le Novère, N., & Hermjakob, H. (2018). BioModels: expanding horizons to include more modelling approaches and formats. *Nucleic Acids Research*, *46*(D1), D1248–D1253. https://doi.org/10.1093/nar/gkx1023

Godin, C., & Caraglio, Y. (1998). A Multiscale Model of Plant Topological Structures. *Journal of Theoretical Biology*, *191*(1), 1–46. https://doi.org/10.1006/jtbi.1997.0561

Guba, E. G., & Lincoln, T. S. (1994). Competing paradigms in qualitative research. In N. K. Denzin & Y. S. Lincoln (Eds.), Handbook of qualitative research (pp. 105-117). *Thousand Oaks, CA: Sage*, 105–117.

Hammer, G. (1998). Crop modelling: Current status and opportunities to advance. *Acta Horticulturae*, *456*(March 1998), 27–36. https://doi.org/10.17660/ActaHortic.1998.456.1

Hammer, G., Messina, C., Wu, A., & Cooper, M. (2019). *Opinion Biological reality and parsimony in crop models — why we need both in crop improvement !* 1–21. https://doi.org/10.1093/insilicoplants/diz010

Hardebolle, C. (2008). *Composition de modèles pour la modélisation multi-paradigme du comportement des systèmes* PhD thesis, Université Paris-Sud XI.

Hatzi, O., Vrakas, D., Nikolaidou, M., Bassiliades, N., Anagnostopoulos, D., & Vlahavas, I. (2012). An Integrated Approach to Automated Semantic Web Service Composition through Planning. *IEEE*

Transactions on Services Computing, *5*(3), 319–332. https://doi.org/10.1109/TSC.2011.20

He, J., Le Gouis, J., Stratonovitch, P., Allard, V., Gaju, O., Heumez, E., Orford, S., Griffiths, S., Snape, J. W., Foulkes, M. J., Semenov, M. A., & Martre, P. (2012). Simulation of environmental and genotypic variations of final leaf number and anthesis date for wheat. *European Journal of Agronomy*, *42*, 22–33. https://doi.org/10.1016/j.eja.2011.11.002

Hinsen, K. (2016). Scientific notations for the digital era. *Physics and Society*, 1–27.

Holzworth, D. P., Huth, N. I., & de Voil, P. G. (2010). Simplifying environmental model reuse. *Environmental Modelling and Software*, *25*(2), 269–275. https://doi.org/10.1016/j.envsoft.2008.10.018

Holzworth, D. P., Huth, N. I., & deVoil, P. G. (2011). Simple software processes and tests improve the reliability and usefulness of a model. *Environmental Modelling and Software*, *26*(4), 510–516. https://doi.org/10.1016/j.envsoft.2010.10.014

Holzworth, D. P., Huth, N. I., DeVoil, P. G., Zurcher, E. J., Herrmann, N. I., McLean, G., Chenu, K., van Oosterom, E. J., Snow, V., Murphy, C., Moore, A. D., Brown, H., Whish, J. P. M., Verrall, S., Fainges, J., Bell, L. W., Peake, A. S., Poulton, P. L., Hochman, Z., … Keating, B. A. (2014a). APSIM - Evolution towards a new generation of agricultural systems simulation. *Environmental Modelling and Software*, *62*, 327–350. https://doi.org/10.1016/j.envsoft.2014.07.009

Holzworth, D. P., Huth, N. I., Fainges, J., Brown, H., Zurcher, E., Cichota, R., Verrall, S., Herrmann, N. I., Zheng, B., & Snow, V. (2018). APSIM Next Generation: Overcoming challenges in modernising a farming systems model. *Environmental Modelling & Software*, *103*, 43–51. https://doi.org/10.1016/j.envsoft.2018.02.002

Holzworth, D. P., Snow, V., Janssen, S., Athanasiadis, I. N., Donatelli, M., Hoogenboom, G., White, J. W., & Thorburn, P. (2014b). Agricultural production systems modelling and software: Current status and future prospects. *Environmental Modelling and Software*, *72*, 276–286. https://doi.org/10.1016/j.envsoft.2014.12.013

Hoogenboom, G., Porter, C. H., Boote, K. J., Shelia, V., Wilkens, P. W., Singh, U., White, J. W., Asseng, S., Lizaso, J. I., Moreno, L. P., Pavan, W., Ogoshi, R., Hunt, L. A., Tsuji, G. Y., & Jones, J. W. (2019). The DSSAT crop modeling ecosystem. In K. J. Boote (Ed.), *Advances in Crop Modeling for a Sustainable Agriculture* (pp. 173–216). Burleigh Dodds Science Publishing, Cambridge, United Kingdom. https://doi.org/10.19103/AS.2019.0061.10

Huang, Y. (2013). *Automated Simulation Model Generation*. Ph.D. Dissertation. Delft University of Technology, The Netherlands.

Hucka, M., Bergmann, F., Hoops, S., Keating, S., Sahle, S., Schaff J. Smith, L., & Wilkinson, D. (2010). *The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core*. Journal of Integrative Bioinformatics,12(2), 266.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., Cuellar, A. A., Dronov, S., Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hedley, W. J., Hodgman, T. C., Hofmeyr, J. H., … Wang, J. (2003). The systems biology markup language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics*, *19*(4), 524–531. https://doi.org/10.1093/bioinformatics/btg015

Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys*, *28*(4), 196-es. https://doi.org/10.1145/242224.242477

Janssen, S. J. C., Porter, C. H., Moore, A. D., Athanasiadis, I. N., Foster, I., Jones, J. W., & Antle, J. M.

(2017). Towards a new generation of agricultural system data, models and knowledge products: Information and communication technology. *Agricultural Systems*, *155*, 200–212. https://doi.org/10.1016/j.agsy.2016.09.017

Jiménez-Navajas, L., Pérez-Castillo, R., & Piattini, M. (2020). Reverse engineering of quantum programs toward kdm models. *Communications in Computer and Information Science*, *1266 CCIS*, 249–262. https://doi.org/10.1007/978-3-030-58793-2_20

Jones, J. W., Antle, J. M., Basso, B., Boote, K. J., Conant, R. T., Foster, I., Godfray, H. C. J., Herrero, M., Howitt, R. E., Janssen, S., Keating, B. A., Munoz-Carpena, R., Porter, C. H., Rosenzweig, C., & Wheeler, T. R. (2017). Brief history of agricultural systems modeling. *Agricultural Systems*, *155*(June), 240–254. https://doi.org/10.1016/j.agsy.2016.05.014

Jones, J. W., Hoogenboom, G., Porter, C. H., Boote, K. J., Batchelor, W. D., Hunt, L. A., Wilkens, P. W., Singh, U., Gijsman, A. J., & Ritchie, J. T. (2003). The DSSAT cropping system model. In *European Journal of Agronomy* (Vol. 18, Issues 3–4). https://doi.org/10.1016/S1161-0301(02)00107-7

Jones, J. W., Keating, B. A., & Porter, C. H. (2001). Approaches to modular model development. *Agricultural Systems*, *70*(2–3), 421–443. https://doi.org/10.1016/S0308-521X(01)00054-3

Jones, J. W., & Luyten, J. C. (1998). Simulation of Biological Processes. In *Agricultural Systems Modeling and Simulation* (Peart, R.), pp. 19–62. The University of Florida.

Jonquet, C., Toulet, A., Arnaud, E., Aubin, S., Dzalé Yeumo, E., Emonet, V., Graybeal, J., Laporte, M. A., Musen, M. A., Pesce, V., & Larmande, P. (2018). AgroPortal: A vocabulary and ontology repository for agronomy. *Computers and Electronics in Agriculture*, *144*, 126–143. https://doi.org/10.1016/j.compag.2017.10.012

Jouault, F., & Kurtev, I. (2006). Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*.Vol. 3844 (pp. 128–138). https://doi.org/10.1007/11663430_14

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J., & Varró, D. (2019). Survey and classification of model transformation tools. *Software & Systems Modeling*, *18*(4), 2361–2397. https://doi.org/10.1007/s10270-018-0665-6

Keating, B., Carberry, P., Hammer, G., Probert, M., Robertson, M., Holzworth, D. P., Huth, N. I., Hargreaves, J. N. G., Meinke, H., Hochman, Z., McLean, G., Verburg, K., Snow, V., Dimes, J. P., Silburn, M., Wang, E., Brown, S., Bristow, K. L., Asseng, S., … Smith, C. J. (2003). An overview of APSIM, a model designed for farming systems simulation. *European Journal of Agronomy*, *18*(3–4), 267–288. https://doi.org/10.1016/S1161-0301(02)00108-9

Keller, R., & Dungan, J. (1999). Meta-modeling: A knowledge-based approach to facilitating process model construction and reuse. *Ecological Modelling*, *119*(2–3), 89–116. https://doi.org/10.1016/S0304-3800(98)00197-5

Kersebaum, K. C., Wallor, E., Lorenz, K., Beaudoin, N., Constantin, J., & Wendroth, O. (2019). *Modeling Cropping Systems with HERMES-Model Capability, Deficits and Data Requirements* (pp. 103–126). https://doi.org/10.2134/advagricsystmodel8.2017.0005

Kim, S. D. (2009). Software Reusability. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc. https://doi.org/10.1002/9780470050118.ecse397

Kleppe, A., Warmer, J., & Bast, W. (2003). MDA Explained: The Model Driven Architectur: Practice and Promise. In *Addison Wesley*. http://link.springer.com/10.1007/s10270-004-0061-2

Klir, G. (2001). *Facets of systems science Second Edition*. Springer Science+Business Media, LLC.

Kluyver, T., Ragan-kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., & Willing, C. (2016). Jupyter Notebooks—a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys*, *24*(2), 131–183. https://doi.org/10.1145/130844.130856

Kuijpers, W. J. P., van de Molengraft, M. J. G., van Mourik, S., van 't Ooster, A., Hemming, S., & van Henten, E. J. (2019). Model selection with a common structure: Tomato crop growth models. *Biosystems Engineering*, *187*, 247–257. https://doi.org/10.1016/j.biosystemseng.2019.09.010

Kulkarni, R., Chavan, A., & Hardikar, A. (2015). Transpiler and it ' s Advantages. *International Journal of Computer Science and Information Technologies (IJCSIT)*, *6*(2), 1629–1631. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.735.4640&rep=rep1&type=pdf

Kurtev, I., Bézivin, J., Jouault, F., & Valduriez, P. (2006). Model-based DSL frameworks. *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA '06*, *2006*, 602. https://doi.org/10.1145/1176617.1176632

Lachaux, M.-A., Roziere, B., Chanussot, L., & Lample, G. (2020). *Unsupervised Translation of Programming Languages*. https://docs.python.org/2/library/2to3.html

Lamanda, N., Roux, S., Delmotte, S., Merot, A., Rapidel, B., Adam, M., & Wery, J. (2012). A protocol for the conceptualisation of an agro-ecosystem to guide data acquisition and analysis and expert knowledge integration. *European Journal of Agronomy*, *38*(1), 104–116. https://doi.org/10.1016/j.eja.2011.07.004

Lamprecht, A.-L., Garcia, L., Kuzak, M., Martinez, C., Arcila, R., Martin Del Pico, E., Dominguez Del Angel, V., van de Sandt, S., Ison, J., Martinez, P. A., McQuilton, P., Valencia, A., Harrow, J., Psomopoulos, F., Gelpi, J. L., Chue Hong, N., Goble, C., & Capella-Gutierrez, S. (2019). Towards FAIR principles for research software. *Data Science*, *3*(1), 37–59. https://doi.org/10.3233/ds-190026

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

Lara, R., Cantador, I., & Castells, P. (2006). Advances in Conceptual Modeling - Theory and Practice. In *Lecture Notes in Computer Science* (Vol. 4231). https://doi.org/10.1007/11908883

Le Franc, Y., Davison, A. P., Gleeson, P., Imam, F. T., Kriener, B., Larson, S. D., Ray, S., Schwabe, L., Hill, S., & De Schutter, E. (2012). Computational Neuroscience Ontology: a new tool to provide semantic meaning to your models. *BMC Neuroscience*, *13*, P149. https://doi.org/10.1186/1471-2202-13-S1-P149

Le Novere, N. (2006). BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic Acids Research*, *34*(90001), D689–D691. https://doi.org/10.1093/nar/gkj092

Ley, P. (1972). *Quantitative Aspects of Psychological Assessment: Introduction*. London: Duckworth.

Li, H., van Katwijk, J., & Levy, A. M. (1992). Reuse of software design and software architecture. *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, 170–177. https://doi.org/10.1109/seke.1992.227932

Lindenmayer, A. (1968). Mathematical Models for Cellular Interactions in Development II. Simple and Branching Filaments with Two-sided Inputs. In *J. Theoret. Biol* (Vol. 18), pp. 280-315.

Linge, S., & Langtangen, H. P. (2016). *Programming for Computations - Python* (Vol. 15). Springer International Publishing. https://doi.org/10.1007/978-3-319-32428-9

Liu, S., Martre, P., Buis, S., Abichou, M., Andrieu, B., & Baret, F. (2019). Estimation of plant and canopy architectural traits using the digital plant phenotyping platform. *Plant Physiology*, *181*(3), 881–890. https://doi.org/10.1104/pp.19.00554

Lloyd, W., David, O., Ascough, J. C., Rojas, K. W., Carlson, J. R., Leavesley, G. H., Krause, P., Green, T. R., & Ahuja, L. R. (2011). Environmental modeling framework invasiveness: Analysis and implications. *Environmental Modelling and Software*, *26*(10), 1240–1250. https://doi.org/10.1016/j.envsoft.2011.03.011

Lobell, D. B., & Asseng, S. (2017). Comparing estimates of climate change impacts from process-based and statistical crop models. *Environmental Research Letters*, *12*(1). https://doi.org/10.1088/1748-9326/aa518a

Mahmood, I., Ayani, R., Vlassov, V., & Moradi, F. (2012). Verifying Dynamic Semantic Composability of BOM-Based Composed Models Using Colored Petri Nets. *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, 250–257. https://doi.org/10.1109/PADS.2012.48

Manceau, L., & Martre, P. (2018). *SiriusQuality-BioMa-Phenology-Component*. https://doi.org/10.5281/ZENODO.2478791

Marshall-Colon, A., Long, S. P., Allen, D. K., Allen, G., Beard, D. A., Benes, B., Von Caemmerer, S., Christensen, A. J., Cox, D. J., Hart, J. C., Hirst, P. M., Kannan, K., Katz, D. S., Lynch, J. P., Millar, A. J., Panneerselvam, B., Price, N. D., Prusinkiewicz, P., Raila, D., … Zhu, X. G. (2017). Crops in silico: Generating virtual crops using an integrative and multi-scale modeling platform. *Frontiers in Plant Science*, *8*, 1–7. https://doi.org/10.3389/fpls.2017.00786

Martins, P., Saraiva, J., Fernandes, J. P., & Van Wyk, E. (2014). Generating attribute grammar-based bidirectional transformations from rewrite rules. *PEPM 2014 - Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Co-Located with POPL 2014*, 63–70. https://doi.org/10.1145/2543728.2543745

Martre, P., Jamieson, P. D., Semenov, M. A., Zyskowski, R. F., Porter, J. R., & Triboi, E. (2006). Modelling protein content and composition in relation to crop nitrogen dynamics for wheat. *European Journal of Agronomy*, *25*(2), 138–154. https://doi.org/10.1016/j.eja.2006.04.007

Martre, P., Marcello, D., Pradal, C., Enders, A., Midingoyi, C. A., Athanasiadis, I., Fumagalli, D., Holzworth, D. P., Hoogenboom, G., Porter, C., Raynal, H., Rizzoli, A. E., & Thorburn, P. (2018). The agricultural model exchange initiative. In IICA (Ed.), *7th AgMIP Global Workshop*.

Masseroli, M., Kaitoua, A., Pinoli, P., & Ceri, S. (2016). Modeling and interoperability of heterogeneous genomic big data for integrative processing and querying. *Methods*, *111*, 3–11. https://doi.org/10.1016/j.ymeth.2016.09.002

Matteis, L., Chibon, P. Y., Espinosa, H., Skofic, M., Finkers, R., Bruskiewich, R., Hyman, G., & Arnaud, E. (2013). Crop ontology: Vocabulary for crop-related concepts. In: P. Larmande, E. Arnaud, I. Mougenot, C. Jonquet, T. Libourel, M. Ruiz (Eds.), 1st International Workshop on Semantics for Biodiversity, CEUR Workshop Proceedings, 979, 37-45

McCarthy, C. (1996). When You Know It, and I Know It, What Is It We Know? Pragmatic Realism and the Epistemologically Absolute. *Philosophy of Education*, 21–29. https://educationjournal.web.illinois.edu/archive/index.php/pes/issue/view/22.html

Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, *152*(1–2), 125–142. https://doi.org/10.1016/j.entcs.2005.10.021

Mens, T., Van Gorp, P., Varró, D., & Karsai, G. (2006). Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science*, *152*(1–2), 143–159. https://doi.org/10.1016/j.entcs.2005.10.022

Mens, T., Van Straeten, R. Der, & D'Hondt, M. (2006). Detecting and resolving model inconsistencies using transformation dependency analysis. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *4199 LNCS*, 200–214. https://doi.org/10.1007/11880240_15

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, *37*(4), 316–344. https://doi.org/10.1145/1118890.1118892

Midingoyi, C. A., Pradal, C., Athanasiadis, I. N., Donatelli, M., Enders, A., Fumagalli, D., Garcia, F., Holzworth, D. P., Hoogenboom, G., Porter, C., Raynal, H., Thorburn, P., & Martre, P. (2020). Reuse of process-based models: automatic transformation into many programming languages and simulation platforms. *In Silico Plants*, *2(1)*. https://doi.org/10.1093/insilicoplants/diaa007

Midingoyi, C. A., Pradal, C., Enders, A., Donatelli, M., Fumagalli, D., Athanasiadis, I., Garcia, F., Holzworth, D. P., Hoogenboom, G., Porter, C., Raynal, H., Thorburn, P., & Martre, P. (2020). *Crop2ML: The centralized framework for crop model component exchange and reuse*. https://doi.org/10.5281/ZENODO.3911713

Misse-chanabier, P., Aranega, V., Polito, G., & Ducasse, S. (2019). Illicium A modular transpilation toolchain from Pharo to C. *IWST19 - International Workshop on Smalltalk Technologies*, 1–12. https://hal.archives-ouvertes.fr/hal-02297860

Monteith, J. L. (1996). The quest for balance in crop modeling. *Agronomy Journal*, *88*(5), 695–697. https://doi.org/10.2134/agronj1996.00021962008800050003x

Mosses, P. D. (2006). Formal Semantics of Programming Languages. *Electronic Notes in Theoretical Computer Science*, *148*(1), 41–73. https://doi.org/10.1016/j.entcs.2005.12.012

Muetzelfeldt, R. (2004). Position paper on declarative modelling in ecological and environmental research. Office for Official Publications of the European Communities, Luxembourg, 88 p. ISBN 92-894-5212-9.

Muetzelfeldt, R., & Massheder, J. (2003). The Simile visual modelling environment. *European Journal of Agronomy*, *18*(3–4), 345–358. https://doi.org/10.1016/S1161-0301(02)00112-0

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

Murthy, V. R. K. (2004). Crop growth modelling and its applications in agricultural meteorology. In *Satellite Remote Sensing and GIS Applications in Agricultural Meteorology* (pp. 235–261). http://www.wamis.org/agm/pubs/agm8/Paper-12.pdf

Netedu, A., Buraga, S. C., Diac, P., & Ţucăr, L. (2020). A Web Service Composition Method Based on OpenAPI Semantic Annotations. *Chao KM., Jiang L., Hussain O., Ma SP., Fei X. (Eds) Advances in E-Business Engineering for Ubiquitous Computing. ICEBE 2019. Lecture Notes on Data Engineering and Communications Technologies, Vol 41. Springer, Cham.*, 342–357. https://doi.org/10.1007/978-3-030-34986-8_25

Neveu, P., Tireau, A., Hilgert, N., Nègre, V., Mineau-Cesari, J., Brichet, N., Chapuis, R., Sanchez, I., Pommier, C., Charnomordic, B., Tardieu, F., & Cabrera-Bosquet, L. (2018). Dealing with multi-source and multi-scale information in plant phenomics: the ontology-driven Phenotyping Hybrid Information System. *New Phytologist*, *221*(1), 588–601. https://doi.org/10.1111/nph.15385

Nunnari, F., & Heloir, A. (2018). *Write-once, transpile-everywhere: re-using motion controllers of virtual humans across multiple game engines*. In: De Paolis L, Bourdot P, eds. Augmented reality, virtual reality, and computer graphics, Vol. 10850. AVR 2018. Lecture Notes in Computer Science. Cham: Springer. doi:10.1007/978-3-319-95270-3_37.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., & Nakamura, S. (2015). Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 574–584. https://doi.org/10.1109/ASE.2015.36

OMG. (2007). *System Modeling Language (SysML)*. Http://Www.Omgsysml.Org/.

Page, E. H. (1994). *Simulation Modeling Methodology: Principles and Etiology of Decision Support*. Virigina Polytechnic Institute and State University.

Palosuo, T., Kersebaum, K. C., Angulo, C., Hlavinka, P., Moriondo, M., Olesen, J. E., Patil, R. H., Ruget, F., Rumbaur, C., Takáč, J., Trnka, M., Bindi, M., Çaldaĝ, B., Ewert, F., Ferrise, R., Mirschel, W., Şaylan, L., Šiška, B., & Rötter, R. (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), 103–114. https://doi.org/10.1016/j.eja.2011.05.001

Papajorgji, P., Clark, R., & Jallas, E. (2009). *The Model Driven Architecture Approach: A Framework for Developing Complex Agricultural Systems* (pp. 1–18). https://doi.org/10.1007/978-0-387-75181-8_1

Parigot, D., & Courbis, C. (2006). *Domain-Driven Development: the SmartTools Software Factory*. INRIA. https://hal.inria.fr/inria-00070419

Parr, T. (2013). The Definite ANTLR 4 Reference. In: *The Pragmatic Programmers*. https://doi.org/10.1016/j.anbehav.2003.06.004

Plaisted, D. A. (2003). *An Abstract Programming System*. 1–34. http://arxiv.org/abs/cs/0306028

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

Pohl, K., Böckle, G., & van der Linden, F. (2005). Software Product Line Engineering, 467 pp. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-28901-1

Porter, C. H., Villalobos, C., Holzworth, D. P., Nelson, R., White, J. W., Athanasiadis, I. N., Janssen, S., Ripoche, D., Cufi, J., Raes, D., Zhang, M., Knapen, R., Sahajpal, R., Boote, K., & Jones, J. W. (2014). Harmonization and translation of crop modeling data to ensure interoperability. *Environmental Modelling and Software*, *62*, 495–508. https://doi.org/10.1016/j.envsoft.2014.09.004

Poskitt, C. M., Dodds, M., Paige, R. F., & Rensink, A. (2014). Towards rigorously faking bidirectional model transformations. *In: Dingel et al. (eds.), AMT 2014. CEUR Workshop Proceedings*, *1277*, 70–75.

Pradal, C. (2019). *Architecture de Dataflow pour des systèmes modulaires et génériques de simulation de plante*. Ph.D. Thesis, Université de Montpellier.

Pradal, C., Dufour-Kowalski, S., Boudon, F., Fournier, C., & Godin, C. (2008). OpenAlea: A visual

programming and component-based software platform for plant modelling. *Functional Plant Biology*, *35*(10), 751–760. https://doi.org/10.1071/FP08084

Pradal, C., Fournier, C., Valduriez, P., & Cohen-Boulakia, S. (2015). OpenAlea: Scientific Workflows Combining Data Analysis and Simulation. *Proceedings of the 27th International Conference on Scientific and Statistical Database Management* (SSDBM' 15). Association for Computing Machinery, New York, USA, 1–6. https://doi.org/10.1145/2791347.2791365

Pradal, C., Varoquaux, G., & Langtangen, H. P. (2013). Publishing scientific software matters. *Journal of Computational Science*, *4*(5), 311–312. https://doi.org/10.1016/j.jocs.2013.08.001

Quesnel, G., Duboz, R., & Ramat, É. (2009). The Virtual Laboratory Environment - An operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, *17*(4), 641–653. https://doi.org/10.1016/j.simpat.2008.11.003

Quinlan, D., & Liao, C. (2011). The ROSE Source-to-Source Compiler Infrastructure. *Cetus Users and Compiler Infrastructure Workshop, in Conjunction with PACT*, 1–3.

R Core Team. (2017). *R: A Language and Environment for Statistical Computing*. https://www.r-project.org/

Ramírez, A., Rodríguez, F., Berenguel, M., & Heuvelink, E. (2004). Calibration and validation of complex and simplified tomato growth models for control purposes in the Southeast of Spain. *Acta Horticulturae*, *654*, 147–154. https://doi.org/10.17660/ActaHortic.2004.654.15

Refsgaard, J. C., van der Sluijs, J. P., Højberg, A. L., & Vanrolleghem, P. A. (2007). Uncertainty in the environmental modelling process - A framework and guidance. *Environmental Modelling and Software*, *22*(11), 1543–1556. https://doi.org/10.1016/j.envsoft.2007.02.004

Richmond, B. (2001). An Introduction to Systems Thinking. *High Performance Systems*. www.iseesystems.com

Richmond, B. M. (1985). STELLA: Software for Bringing System Dynamics to the Other 98%. *Proceedings of the 1985 International System Dynamics Conference*, 706–718.

Rico, D. F., Sayani, H. H., & Field, R. F. (2008). History of Computers, Electronic Commerce and Agile Methods. In *Advances in Computers* (Vol. 73, Issue 08), 1-55. Elsevier Masson SAS. https://doi.org/10.1016/S0065-2458(08)00401-4

Ritchie, J. T., & Otter, S. (1985). Description and performance of CERES-Wheat: a user-oriented wheat yield model. *United States Department of Agriculture, Agricultural Research Services, ARS*, *38,* 159–175.

Rizzoli, A. E., Donatelli, M., Athanasiadis, I. N., Villa, F., & Huber, D. (2008). Semantic links in integrated modelling frameworks. *Mathematics and Computers in Simulation*, *78*(2–3), 412–423. https://doi.org/10.1016/j.matcom.2008.01.017

Robinson, S., Nance, R. E., Paul, R. J., Pidd, M., & Taylor, S. J. E. (2004). Simulation model reuse: Definitions, benefits and obstacles. *Simulation Modelling Practice and Theory*, *12*(7-8 SPEC. ISS.), 479–494. https://doi.org/10.1016/j.simpat.2003.11.006

Rosenzweig, C., Jones, J. W., Hatfield, J. L., Ruane, A. C., Boote, K. J., Thorburn, P., Antle, J. M., Nelson, G. C., Porter, C., Janssen, S., Asseng, S., Basso, B., Ewert, F., Wallach, D., Baigorria, G., & Winter, J. M. (2013). The Agricultural Model Intercomparison and Improvement Project (AgMIP): Protocols and pilot studies. *Agricultural and Forest Meteorology*, *170*, 166–182. https://doi.org/10.1016/j.agrformet.2012.09.011

Rötter, R. P., Carter, T. R., Olesen, J. E., & Porter, J. R. (2011). Crop-climate models need an overhaul. *Nature Climate Change*, *1*(4), 175–177. https://doi.org/10.1038/nclimate1152

Roy, A. (2017). *Cookiecutter: Better Project Templates — cookiecutter 1.7.2 documentation*. https://cookiecutter.readthedocs.io/en/1.7.2/

Rusinkiewicz, M., & Sheth, A. (1995). Specification and Execution of Transactional Workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond* (pp. 592–620). ACM Press/Addison-Wesley Publishing Co. https://www.researchgate.net/publication/2623665

Samourkasidis, A., & Athanasiadis, I. N. (2020). A semantic approach for timeseries data fusion. *Computers and Electronics in Agriculture*, *169*, 105171. https://doi.org/10.1016/j.compag.2019.105171

Samourkasidis, A., Papoutsoglou, E., & Athanasiadis, I. N. (2019). A template framework for environmental timeseries data acquisition. *Environmental Modelling and Software*, *117*, 237–249. https://doi.org/10.1016/j.envsoft.2018.10.009

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

Schilstra, M. J., Li, L., Matthews, J., Finney, A., Hucka, M., & Le Novère, N. (2006). CellML2SBML: Conversion of CellML into SBML. *Bioinformatics*, *22*(8), 1018–1020. https://doi.org/10.1093/bioinformatics/btl047

Schmidt, D. C. (2006). Model-Driven Engineering. In *IEEE Computer* (Vol. 39, Issue 2). http://www.computer.org/portal/site/computer/menuitem.e533b16739f5...

Sharpley, A. N., & Williams, J. R. (1990). EPIC: The erosion-productivity impact calculator. *U.S. Department of Agriculture Technical Bulletin*, *1768*, 235. http://agris.fao.org/agris-search/search.do?recordID=US9403696

Singh, A. K. (1994). Crop growth simulation models. IASRI, New Delhi, India, 497-509.

Slonneger, K., & Kurtz, B. L. (1995). *Formal Syntax and Semantics of Programming Languages A Laboratory Based Approach* (Addison-Wesley (ed.)). Addison-Wesley.

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems and Software*, *56*(1), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3

Steduto, P., Hsiao, T. C., Raes, D., & Fereres, E. (2009). Aquacrop-the FAO crop model to simulate yield response to water: I. concepts and underlying principles. *Agronomy Journal*, *101*(3), 426–437. https://doi.org/10.2134/agronj2008.0139s

Stepanov, A., & Lee, M. (1995). *The Standard Template Library*. Tech. Rep. HPL-95-11, HP Labs.

Stöckle, C. O., Donatelli, M., & Nelson, R. (2003). CropSyst, a cropping systems simulation model. *European Journal of Agronomy*, *18*(3–4), 289–307. https://doi.org/10.1016/S1161-0301(02)00109-0

Stöckle, C. O., & Kemanian, A. R. (2020). Can Crop Models Identify Critical Gaps in Genetics, Environment, and Management Interactions? *Frontiers in Plant Science*, *11*. https://doi.org/10.3389/fpls.2020.00737

Szabo, C., & Teo, Y. M. (2009). An Approach for Validation of Semantic Composability in Simulation Models. *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 3–10. https://doi.org/10.1109/PADS.2009.14

Terekhov, A. A., & Verhoef, C. (2000). Realities of language conversions. *IEEE Software*, *17*(6), 111–124. https://doi.org/10.1109/52.895180

Timlin, D., Pachepsky, Y. A., & Acock, B. (1996). A design for a modular, generic soil simulator to interface with plant models. *Agronomy Journal*, *88*(2), 162–169. https://doi.org/10.2134/agronj1996.00021962008800020008x

Tolk, A. (2016). Tutorial on the engineering principles of combat modeling and distributed simulation. *Proceedings - Winter Simulation Conference*, *December 2016*, 255–269. https://doi.org/10.1109/WSC.2016.7822094

Van Delden, H., Seppelt, R., White, R., & Jakeman, A. J. (2011). A methodology for the design and development of integrated models for policy support. *Environmental Modelling and Software*, *26*(3), 266–279. https://doi.org/10.1016/j.envsoft.2010.03.021

Van Evert, F., Holzworth, D. P., Muetzelfeldt, R., Rizzoli, A. E., & Villa, F. (2005). Convergence in integrated modeling frameworks. *MODSIM05 - International Congress on Modelling and Simulation: Advances and Applications for Management and Decision Making, Proceedings*, 745–750.

Van Ittersum, M. K., Leffelaar, P. A., Van Keulen, H., Kropff, M. J., Bastiaans, L., & Goudriaan, J. (2003). On approaches and applications of the Wageningen crop models. *European Journal of Agronomy*, *18*(3–4), 201–234. https://doi.org/10.1016/S1161-0301(02)00106-5

van Kraalingen, D. W. G., Knapen, M. J. R., de Wit, A., & Boogaard, H. L. (2020). *WISS a Java Continuous Simulation Framework for Agro-Ecological Modelling* (pp. 242–248). https://doi.org/10.1007/978-3-030-39815-6_23

Vasenin, V. A., & Krivchikov, M. A. (2020). Intermediate Representation of Programs with Type Specification Based on Pattern Matching. *Programming and Computer Software*, *46*(1), 57–66. https://doi.org/10.1134/S0361768820010077

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

Villa, F., Balbi, S., Athanasiadis, I. N., & Caracciolo, C. (2017). Semantics for interoperability of distributed data and models: Foundations for better-connected information. *F1000Research*, *6*(2), 686. https://doi.org/10.12688/f1000research.11638.1

Villa, F., Donatelli, M., Rizzoli, A. E., Krause, P., Kralisch, S., & Van Evert, F. K. (2006). Declarative modelling for architecture independence and data/model integration: A case study. *Voinov, A., Jakeman, A.J., Rizzoli, A.E. (Eds). Proceedings of the IEMSs Third Biennial Meeting:" Summit on Environmental Modelling and Software". International Environmental Modelling and Software Society, Burlington, USA, July 2006,* 1–6. http://www.iemss.org/iemss2006/sessions/all.html

Von Bertalanffy, L. (1969). General System Theory. Foundations, Development, Applications. In *Science* (Issue 3880). https://doi.org/10.1126/science.164.3880.681

Von Bloh, W., Schaphoff, S., Müller, C., Rolinski, S., Waha, K., & Zaehle, S. (2018). Implementing the nitrogen cycle into the dynamic global vegetation, hydrology, and crop growth model LPJmL (version 5.0). *Geoscientific Model Development*, *11*(7), 2789–2812. https://doi.org/10.5194/gmd-11-2789-2018

Walker, W. E., Harremoës, P., Rotmans, J., van der Sluijs, J. P., van Asselt, M. B. A., Janssen, P. P. M., & Krayer von Kraus, M. P. (2003). Defining uncertainty: a conceptual basis for uncertainty

management in model-based decision-support. *Integrated Assessment*, *4*, 5–17. https://doi.org/10.1076/iaij.4.1.5.16466

Wallach, D., Makowski, D., Jones, J. W., & Brun, F. (2006). Working with Dynamic Crop Models: evaluation, analysis, parameterization and applications, Elsevier, Amsterdam, The Netherlands, 447p

Wang, E., Brown, H. E., Rebetzke, G. J., Zhao, Z., Zheng, B., & Chapman, S. C. (2019). Improving process-based crop models to better capture genotype×environment×management interactions. *Journal of Experimental Botany*, *70*(9), 2389–2401. https://doi.org/10.1093/jxb/erz092

Wang, E., & Engel, T. (2000). SPASS: A generic process-oriented crop model with versatile windows interfaces. *Environmental Modelling and Software*, *15*(2), 179–188. https://doi.org/10.1016/S1364-8152(99)00033-X

Wang, E., Martre, P., Zhao, Z., Ewert, F., Maiorano, A., Rötter, R. P., Kimball, B. A., Ottman, M. J., Wall, G. W., White, J. W., Reynolds, M. P., Alderman, P. D., Aggarwal, P. K., Anothai, J., Basso, B., Biernath, C., Cammarano, D., Challinor, A. J., De Sanctis, G., … Asseng, S. (2017). The uncertainty of crop yield projections is reduced by improved temperature response functions. *Nature Plants*, *3*(July). https://doi.org/10.1038/nplants.2017.102

Washizaki, H., Guéhéneuc, Y. G., & Khomh, F. (2018). ProMeTA: a taxonomy for program metamodels in program reverse engineering. *Empirical Software Engineering*, *23*(4), 2323–2358. https://doi.org/10.1007/s10664-017-9592-3

Wyatt, D. L. (1990). A framework for reusability using graph-based models. *1990 Winter Simulation Conference Proceedings*, 472–476. https://doi.org/10.1109/WSC.1990.129562

Yin, X., & Struik, P. C. (2015). Crop systems biology: Narrowing the gaps between crop modelling and genetics. *Crop Systems Biology: Narrowing the Gaps Between Crop Modelling and Genetics*, *February*, 1–233. https://doi.org/10.1007/978-3-319-20562-5

Yin, X., & van Laar, H. H. (2005). *Crop Systems Dynamics – an Ecophysiological Simulation Model for Genotype-by-environment Interaction.* Wageningen Academic Publishers. https://doi.org/10.3920/978-90-8686-539-0

Yokoyama, T., Axelsen, H. B., & Glück, R. (2008). Principles of a reversible programming language. *Proceedings of the 2008 Conference on Computing Frontiers - CF '08*, 43-54. https://doi.org/10.1145/1366230.1366239

Zeigler, B. P., Muzy, A., & Kofman, E. (2018). Theory of Modeling and Simulation: Discrete Event & Iterative System. In *Theory of Modeling and Simulation*. Elsevier. https://doi.org/10.1016/C2016-0-03987-6

Zhao, C., Liu, B., Xiao, L., Hoogenboom, G., Boote, K. J., Kassie, B. T., Pavan, W., Shelia, V., Kim, K. S., Hernandez-Ochoa, I. M., Wallach, D., Porter, C. H., Stockle, C. O., Zhu, Y., & Asseng, S. (2019). A SIMPLE crop model. *European Journal of Agronomy*, *104*, 97–106. https://doi.org/10.1016/J.EJA.2019.01.009