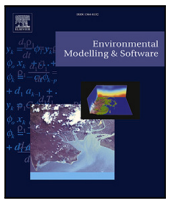
Environmental Modelling and Software 194 (2025) 106661

Contents lists available at ScienceDirect

Environmental Modelling and Software

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



A lightweight dataflow-based software framework for building forest simulators

Tapio Lempinen [a,c,*], Lauri Mehtätalo [a], Annika Kangas [a], Tero Heinonen [c], Paulo Borges [b], Jari Vauhkonen [c]

[a] Natural Resources Institute, Finland, Yliopistokatu 6, Joensuu, 80100, Finland
[b] Natural Resources Institute, Finland, Latokartanonkaari 9, Helsinki, 00790, Finland
[c] School of Forest Sciences, University of Eastern, Finland, Yliopistokatu 7, Joensuu, 80100, Finland

ARTICLE INFO

Keywords:
Simulation
Forest management planning
Model linking
Domain-specific language
Software design

ABSTRACT

Simulators are critical tools for decision-making in forest management. We propose a dataflow-based model linking approach to increase the flexibility and modularity of forest simulator software while maintaining high computational efficiency. Our approach dynamically constructs a data model and model chains for simulation based on a model library, enabled treatments, and requested output variables. Models are framework- and language-independent pure functions, described through metadata in a domain-specific language. A case study with three model libraries demonstrates the applicability and efficiency of our approach. We observed a 96% speedup compared to an unoptimized real-world model implementation, while 95% of our code (measured by lines) was framework-independent and reusable. We observed a 15% slowdown compared to an optimized hand-written C implementation of a simpler model. We conclude that dataflow-based model linking can be used to build flexible and modular simulation software with a small runtime overhead.

1. Introduction

Forest simulators are software tools that use forest dynamics models to predict forest development under hypothetical scenarios (Hasenauer, 2006). Modeling approaches vary between empirical and process-based, spatially dependent and independent, deterministic and stochastic, and tree-level, stand-level and landscape-level (Weiskittel et al., 2011). The level of sophistication of forest dynamics models has historically kept increasing simultaneously with the availability of data and computing resources (Shifley et al., 2017). In addition to forest dynamics, simulators also include models for silvicultural treatments, economics, natural disturbances, and other information of interest such as biodiversity and carbon sequestration (e.g. Hynynen et al., 2002; Lämås et al., 2023).

Forest simulators are widely utilized in decision-making, planning, research and education (Muys et al., 2010; Nobre et al., 2016; Segura et al., 2014) and therefore it is important that they are kept up-to-date with the latest scientific knowledge, as modeling approaches advance and data availability increases. This importance has led to the development of forest simulator software frameworks such as Capsis (Dufour-Kowalski et al., 2011), SIMO (Rasinmäki et al., 2009), and SiTree (Antón-Fernández and Astrup, 2022). Frameworks abstract technical details and provide reusable implementations of common functionality (Gamma et al., 1994), allowing researchers to focus on implementation and integration of models. Forest simulator frameworks are related to generic environmental modeling frameworks (e.g. OMS (David et al., 2013), OpenMI (Harpham et al., 2019; Moore and Tindall, 2005)), which support the reuse and sharing of models regardless of their scientific domain. Although the forestry community has opted to use forestry-specific frameworks instead of generic ones, both types of frameworks share similar goals and designs.

Forest models are typically composed of case-specific sub-models (e.g. per region, soil type, or tree species) for each output and auxiliary variable (e.g. Hynynen et al., 2002; Lämås et al., 2023). Sub-models are developed separately, and therefore it is desirable that not just the composite model, but also its individual components can easily be updated and reused. Sub-models can either be linked directly in code by explicitly programming the calculation logic, or implemented as reusable components in a framework. However, as the number of sub-models and variables increases, defining and managing the links between sub-models becomes more difficult, consuming time and resources, and reducing the maintainability of the simulator system (Nuutinen et al., 2010). The accumulation of complexity over time decreases flexibility (Martin, 2017) and may ultimately lead to the need to start over with a new system (e.g. Salminen et al., 2005; Strîmbu et al., 2023).

* Corresponding author at: Natural Resources Institute, Finland, Yliopistokatu 6, Joensuu, 80100, Finland.
E-mail address: tapio.lempinen@luke.fi (T. Lempinen).

https://doi.org/10.1016/j.envsoft.2025.106661
Received 25 April 2025; Received in revised form 29 July 2025; Accepted 24 August 2025
Available online 11 September 2025

Complexity also increases the likelihood of bugs, which may be hard to find due to the difficulty of testing numerical simulation software (He et al., 2020).

In this paper, we describe a new forest simulator software framework and its underlying dataflow-based model linking scheme. Our work focuses on developing a framework-level solution to the maintainability challenges arising from the complex internal structures of some forest models. To this end, we take a fine-grained, automated approach. In general, the granularity (level of detail) of a component represents a tradeoff between component complexity and linking complexity: a model can be divided into a large number of simple, fine-grained components, or a smaller number of more complex, coarse-grained components (de Kok et al., 2015; Donatelli and Rizzoli, 2008). However, the complexity of linking a large number of fine-grained components can be reduced by linking models automatically based on metadata (e.g. Nuutinen et al., 2010; Villa, 2007 in the context of knowledge representation and reasoning). Model linking and calling in our framework is based on a domain-specific language (DSL) for describing model metadata. The model linking scheme is language-agnostic, and support for multiple programming languages is implemented. While our application is in forestry, the model linking scheme is general and also applicable in other domains. In particular, model linking does not assume specific objects or variables: required state variables are discovered as a part of model linking. This way our scheme can support many of the various representations used by forestry models (e.g. individual trees (Hynynen et al., 2002), strata (Siipilehto, 2000), cells (Wang et al., 2014), matrices (Packalen et al., 2014)). To support the reusability and testability of models, our framework takes a lightweight (Lloyd et al., 2011; Rizzoli et al., 2008) approach: all models are pure functions operating only on numeric values, and do not implement any specific interface or use framework-specific functions or classes.

While coarse-grained components require few model calls that each perform a large amount of work, making the computational overhead of model calling negligible (e.g. Knapen et al., 2013), for fine-grained components, the inverse is true: the system needs to perform a large number of model calls with each performing a small amount of work, meaning the model call overhead has a larger effect on overall runtime. Our framework aims to minimize the computational overhead of dynamic model selection and cross-language function calling by compiling generated model chains to machine code at runtime using a just-in-time (JIT) compiler. We take care to minimize the total number of model function calls by generating model chains that only call a model if its result is known to be needed, and reuse results of previous model calls where possible.

The rest of this paper is organized as follows. Section 2 sets up formal semantics for the model linking scheme, independently of the technical implementation. Section 3 describes the software implementation of the forest simulator framework based on the scheme of Section 2. Section 4 presents a case study with three different model libraries. Sections 5 and 6 conclude by summarizing the results of the case study, limitations of our approach, and potential future work.

2. Model linking scheme

2.1. Concepts

Code in forest simulator software commonly serves one of the three following purposes:

1. pure, self-contained model functions that implement mathematical models such as prediction of tree attributes like crown ratio or volume;
2. calculation logic that combines individual model functions into model chains such as growth or regeneration;
3. application logic that updates simulation state by executing model chains, and reports simulation outputs.

Type 1 functions are simple: often they are just direct transcriptions of mathematical formulas. They are self-contained, single-purpose, and rarely modified. Type 2 code, on the other hand, is more complex because it must take into account the dependencies between all type 1 functions it calls. Every time a new type 1 function is added, type 2 code and related data structures must be modified accordingly.

The goals of our model linking scheme are twofold: (i) separate model functions from application logic and other model functions, so that they can be independently reused and tested; and (ii) automatically generate model chains and a data model from model functions, so that there is no need to write and maintain any type 2 code. We deal with two main concepts: *models* that describe how variables are computed from other variables, and *applications* that describe what to do with those variables. Models do not directly interact with applications or other models. Instead, they refer to variable names from a common *vocabulary*. Given an application and a *model library* consisting of individual models, model linking produces model chains and a data model suitable for simulating the application.

2.2. Semantics

We now set up precise semantics for how the concepts interact with each other. This is important, because the semantics determine the types of optimizations and analyses a software implementation can perform. Our aim is to define relaxed semantics that allow the dataflow compiler freedom in the choice of execution order and elimination of redundant work. Namely, we avoid any procedural constructs such as side-effects and mutation, and instead treat models as pure mathematical functions linking the values of input and output variables.

Formally, a vocabulary is a pair $(\mathcal{V}, \mathcal{T})$, where $\mathcal{V}$ is a set of *variables*, $\mathcal{T}$ is a set of *tables*, and each variable $v \in \mathcal{V}$ belongs to a table $T(v) \in \mathcal{T}$. Tables are analogous to database tables whose columns are variables, and rows are *instances*. A *simulation state* s assigns a value $s(v, i)$ for each instance $i \in \mathbb{N}$ of a variable $v \in \mathcal{V}$. The number of instances a variable has is given by the *size* of its table, which is a *global* variable. The table of global variables $g \in \mathcal{T}$ always has size one. We use the notation $s(v)$ to refer to the values of all instances of v. If z is a list of n integers, $s(v, z)$ refers to the list of values $s(v, z_i)$ for each $i = 1, \dots, n$.

An application is a pair $(\mathcal{A}, \mathcal{O})$ consisting of a set of *actions* $\mathcal{A}$ and a set of *observed variables* $\mathcal{O} \subset \mathcal{V}$. Actions define how the values of variables change during simulation, and observed variables determine the simulation output. Formally, an action $a \in \mathcal{A}$ maps a state s to another state $s' = a(s)$. We denote by $O(a)$ the set of variables a may change, and by $I(a)$ the set of variables a depends on. A *simulation* of n steps starts from an initial state s_0, applies actions $a_i \in \mathcal{A}$ at each $i = 1, \dots, n$ to obtain $s_i = a_i(s_{i-1})$, and observes $s_i(v)$ for $v \in \mathcal{O}$.

A model library $\mathcal{M}$ is a set of models. A model $m \in \mathcal{M}$ is an object with: a table $T(m) \in \mathcal{T}$; an *output* $O(m) \in \mathcal{V}$; *inputs* $I_1(m), \dots, I_{N(m)}(m) \in \mathcal{V}$; input instances $J_1(m), \dots, J_{N(m)}(m) \in \mathcal{V}$; a *condition* $C(m) \in \mathcal{V}$; and a *model function* f_m. Models tie together the values of variables. Each model states: "if the model condition is true, then the value of the output variable is the result of applying f_m on the input variables". Formally, at every simulation step, the state s must satisfy the invariant

$$s(C(m), i) \implies s(O(m), i) = f_m(s(I_1(m), s(J_1(m), i)), \dots, s(I_{N(m)}(m), s(J_{N(m)}(m), i))) \tag{1}$$

for each model $m \in \mathcal{M}$ and instance i. The requirement that each model outputs one variable is technical and simplifies notation, but does not limit generality (see Appendix A.1).

The set of all variables referenced by a model m is denoted by $I(m) = \{I_1(m), \dots, I_{N(m)}(m), J_1(m), \dots, J_{N(m)}(m), C(m), S(T(m))\}$, where $S(\bullet)$ denotes the size variable of a table. We also define $I(v)$ and $O(v)$ for a variable $v \in \mathcal{V}$ as the set of models that have v as an output or input, respectively: $I(v) = \{m \in \mathcal{M} : O(m) = v\}$, and $O(v) = \{m \in$

$\mathcal{M} : v \in I(m)\}$. If $I(v) = \emptyset$, that is, v is not the output variable of any model, then v is called a *data* variable. Otherwise, v is called a *computed* variable. We assume that at each simulation state s, for each computed variable v, the condition $s(C(m), i)$ in (1) is true for exactly one model $m \in I(v)$ and one instance i. This is a technical assumption that can always be satisfied by suitable augmentation of the vocabulary and model library (see Appendix A.1). It guarantees that each computed variable has a well-defined value, and that the simulation state is completely determined by the values of data variables. We can now also assume that all variables $O(a)$ modified by an action a are data variables, because computed variables only have one possible value. To obtain an initial simulation state s_0, it is sufficient to assign an initial value to each data variable, for example by reading it from a database.

The vocabulary $(\mathcal{V}, \mathcal{T})$ is mostly implicit. Namely, given an application $(\mathcal{A}, \mathcal{O})$ and a model library $\mathcal{M}$, we can set

$$\mathcal{V} = \mathcal{O} \cup \mathcal{V}_\mathcal{A} \cup \mathcal{V}_\mathcal{M},$$
$$\mathcal{T} = \{T(x) : x \in \mathcal{V} \cup \mathcal{M}\},$$

where $\mathcal{V}_\mathcal{A}$ and $\mathcal{V}_\mathcal{M}$ are obtained by collecting all variables referenced in the actions and models, respectively:

$$\mathcal{V}_\mathcal{A} = \bigcup_{a \in \mathcal{A}} I(a) \cup O(a),$$
$$\mathcal{V}_\mathcal{M} = \bigcup_{m \in \mathcal{M}} I(m) \cup \{O(m)\}.$$

What remains to be explicitly specified is the size of each table, that is, the variables $S(t)$ for $t \in \mathcal{T}$.

2.3. Dataflow graph analysis

We construct and analyze a *dataflow graph* to determine model chains and a data model. Given a vocabulary $(\mathcal{V}, \mathcal{T})$ and a model library $\mathcal{M}$, the dataflow graph G is a directed graph with nodes $\mathcal{V} \cup \mathcal{M}$, forward-edges O, and back-edges I, where O and I are as defined in the previous section. We denote by $R_E(x)$ the set of nodes reachable through edges $E \in \{I, O\}$ from a node $x \in \mathcal{V} \cup \mathcal{M}$ or an action $x \in \mathcal{A}$ of an application $(\mathcal{A}, \mathcal{O})$.

Evaluating a computed variable $v \in \mathcal{V}$ corresponds to a partial post-order walk of its model chain, the subgraph $R_I(v)$. However, our relaxed semantics give us freedom in choosing the evaluation strategy, as we are not required to call model functions in any certain order or number of times. The main choice to be made is whether the value $s(v)$ of a computed variable should be stored after computing it once, or recomputed every time it is used. Storing the value requires additional bookkeeping during reads and writes, and prevents the dataflow compiler from inlining the expression. It is therefore beneficial to only store the values of expressions which are both expensive to compute and used multiple times. A sufficient (but not necessary) condition to detect variables v whose computed value is only used once is: (i) $|O(v)| = 1$; and (ii) the dependency between instances of $m \in O(v)$ and v is one-to-one; and (iii) $m \in R_O(a) \implies v \in R_O(a)$ for each $a \in \mathcal{A}$. The first two conditions guarantee that only one instance of one model uses the value, and the last condition guarantees that whenever the value of any input variable of the model changes, the value of v also changes.

Typically all models in the model library and all variables in the vocabulary are not required to simulate a given application. We can analyze the dataflow graph to determine a *minimal* set of models and variables. In the extreme case, if $\mathcal{O} = \emptyset$, then the application produces no output, so we do not need to simulate or keep track of anything, regardless of what models are available in the model library. In general, we wish to find minimal a minimal data model $\widehat{\mathcal{V}} \subset \mathcal{V}$ and set of models $\widehat{\mathcal{M}} \subset \mathcal{M}$ such that we can produce the desired output $\mathcal{O}$. Clearly we must have at least $\mathcal{O} \subset \widehat{\mathcal{V}}$. We also must include any models and variables that are needed for computing the variables we are interested in. That is, for all $x \in \widehat{\mathcal{V}} \cup \widehat{\mathcal{M}}$ we need $I(x) \subset \widehat{\mathcal{V}} \cup \widehat{\mathcal{M}}$. We must also include any variables required for the application's actions. However, since the results of actions are not observed directly, but only through the observation set $\mathcal{O}$, we do not need to perform every action $\mathcal{A}$. Instead, it is sufficient to perform only those actions which affect the values of variables in $\widehat{\mathcal{V}}$, that is, the actions $\widehat{\mathcal{A}} = \{a \in \mathcal{A} : O(a) \cap \widehat{\mathcal{V}} \neq \emptyset\}$. Therefore we require that $I(a) \subset \widehat{\mathcal{V}}$ for each $a \in \widehat{\mathcal{A}}$. Putting everything together, $\widehat{\mathcal{V}}$ and $\widehat{\mathcal{M}}$ can be found as the solution of the dataflow system

$$\begin{aligned} &\mathcal{O} \subset \widehat{\mathcal{V}}, \\ &I(x) \subset \widehat{\mathcal{V}} \cup \widehat{\mathcal{M}} \qquad \forall x \in \widehat{\mathcal{V}} \cup \widehat{\mathcal{M}}, \\ &I(a) \subset \widehat{\mathcal{V}} \qquad \forall a \in \widehat{\mathcal{A}}, \\ &\widehat{\mathcal{A}} = \{a \in \mathcal{A} : O(a) \cap \widehat{\mathcal{V}} \neq \emptyset\}. \end{aligned} \tag{2}$$

System (2) is the core of the model linking scheme. It is what allows us to dynamically determine the data model $\widehat{\mathcal{V}}$, models $\widehat{\mathcal{M}}$, and actions $\widehat{\mathcal{A}}$ on a per-simulation basis without explicit configuration or logic. It also provides flexibility: augmenting system (2) with application-specific rules enables behavior that is generic over the available variables, as we will demonstrate in Section 4. System (2) is readily solved by an iterative fixpoint algorithm. That is, we start at $\widehat{\mathcal{V}}_0 = \mathcal{O}$, $\widehat{\mathcal{M}}_0 = \emptyset$ and keep adding variables, models, and actions according to the rules until the sets no longer change.

2.4. Illustrative example

We consider the growth models of Nyyssönen and Mielikäinen (1978):

$$\begin{aligned} d_5 = (1 + 0.01 \exp(&5.4625 - 0.6675 \log T - 0.4758 \log B + 0.1173 \log D \\ &- 0.9442 \log H_{\text{dom}} - 0.3631 \log d + 0.7762 \log h))^5 d \quad \text{if } s = 1, \end{aligned} \tag{3a}$$

$$\begin{aligned} d_5 = (1 + 0.01 \exp(&6.9342 - 0.8808 \log T - 0.4982 \log B + 0.4159 \log D \\ &- 0.3865 \log H_{\text{med}} - 0.6267 \log d + 0.1286 \log h))^5 d \quad \text{otherwise}, \end{aligned} \tag{3b}$$

and the height-diameter (H-D) models of Siipilehto (2000):

$$h = 1.3 + d^2/(0.894 + 0.185d)^2 \quad \text{if } s = 1, \tag{4a}$$

$$h = 1.3 + d^3/(1.811 + 0.308d)^3 \quad \text{if } s = 2, \tag{4b}$$

$$h = 1.3 + d^2/(0.898 + 0.242d)^2 \quad \text{if } s = 3, \tag{4c}$$

where h, d, d_5, s, and t are the height (m), diameter (cm), diameter after 5 years (cm), species (categorical 1–3), and age (years) of a tree respectively, and T, B, D, H_{med}, and H_{dom} are the mean age (years), total basal area m^2/ha, mean diameter (cm), mean height (m), and dominant height (m) of the stand, respectively. The vocabulary is given by

$$\mathcal{T} = \{\texttt{tree}, \texttt{stand}, \texttt{global}\},$$
$$\mathcal{V} = \{h, d, d_5, s, t, T, B, D, H_{\text{med}}, H_{\text{dom}}\} \cup \mathcal{V}_{\text{technical}},$$

with $T(\bullet) = \texttt{tree}$ for tree-level variables and $T(\bullet) = \texttt{stand}$ for stand-level variables. Technical variables $\mathcal{V}_{\text{technical}}$ include table sizes and model conditions ("$s = 1, 2, 3$"), which we omit for readability. The model library contains the models (3a)–(3b) and (4a)–(4c), as well as formulas for the computed variables T, B, H_{med}, H_{dom}. We denote the latter by f_v, where v is the computed variable. Therefore, we have

$$\mathcal{M} = \{\text{(3a)}, \text{(3b)}, \text{(4a)}, \text{(4b)}, \text{(4c)}, f_T, f_B, f_{H_{\text{med}}}, f_{H_{\text{dom}}}\} \cup \mathcal{M}_{\text{technical}},$$

with $T(\bullet) = \texttt{tree}$ for (3a)–(3b), (4a)–(4c), and $T(f_\bullet) = \texttt{stand}$. Technical models $\mathcal{M}_{\text{technical}}$ include models for technical variables, and are omitted for readability. From the dataflow graph (Fig. 1) we can identify nodes without incoming edges (d, s, and t) as the data variables. Any other variable can be computed from these variables.

We now consider an application with one action: a 5-year growth step g_5. The growth action replaces d with d_5 and t with $t + 5$, that is, $s'(d_5) = s(d)$ and $s'(t) = s(t) + 5$ for a state $s' = g_5(s)$. The application observes one variable: the standing basal area B. Formally, we have

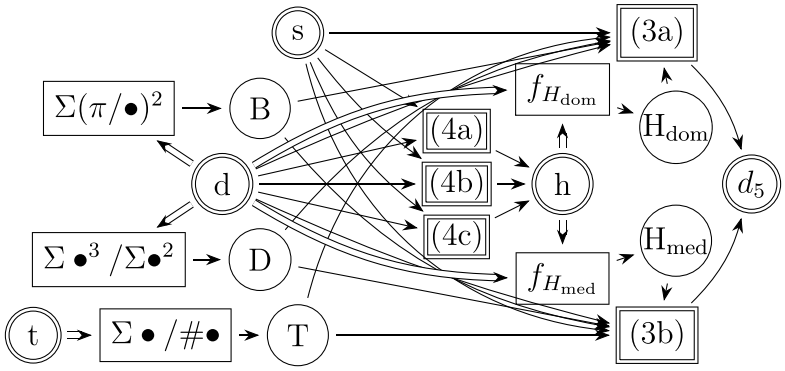


Fig. 1. A dataflow graph. Models are squared (□), variables are circled (○). Stand-level nodes have a single border (○), tree-level nodes have a double border (◎). Regular lines (⟶) select the same instance, double lines (⟹) select all instances.

$\mathcal{A} = \{g_5\}$ and $\mathcal{O} = \{B\}$. We determine the models and variables required to simulate the action by solving (2). We start from $\hat{\mathcal{V}}_0 = \{B\}$. To compute B, we also need f_B and d. Action g_5 modifies d, so we need its input, d_5. For d_5, we need the whole graph, so we have found a fixpoint. If the graph included other models, for example biomass or volume, they would not be needed for the simulation unless the application observed them. If the application observed only a variable not affected by growth, for example species, then simulation could skip executing g_5.

3. Software implementation

3.1. Architecture

The main components of the software (Fig. 2) are a *dataflow engine*, responsible for dealing with model libraries, and a *simulation engine* that wraps the dataflow engine and other internal components behind a Lua scripting API. Applications must use the Lua API, and are tightly coupled to the framework, while model functions are independent of the framework, applications, and other model functions, and can be shared among multiple applications or reused and tested outside the framework. Model functions operate with numeric inputs and outputs, rather than framework-specific data structures. Model libraries are described to the dataflow engine using a domain-specific language (DSL), and can be implemented using any combination of programming languages supported by the dataflow engine (currently R, Lua, and any language that can produce a C ABI dynamic library, e.g. C/C++, Fortran, Rust). The framework is generic and can also be used in non-forestry contexts: forestry simulators are simply created by using a model library containing forestry models and an application performing forestry actions.

The goal of creating a hard divide between the model library and the application is to maximize the relative amount of simple, reusable code compared to complex, framework-specific code. Ideally, most code lives in model libraries and applications are kept as small as possible. Model functions rarely need to change, and existing, tested implementations can be reused because there is no language- or framework-specific interface. Model functions cannot mutate the simulation state, which eliminates the possibility of mutation-related programming errors (Martin, 2017), and benefits code comprehensibility (Dolado et al., 2003). Automatic generation of model chains and the data model leaves the (ideally small) applications as the only potential source of mutation-related bugs and complexity issues.

3.2. DSL

The following is a brief overview of the DSL syntax and semantics. A full syntax listing is provided in Appendix A.2. The DSL consists of a sequence of top-level definitions, of which the most important one is the `model` definition. The `model` keyword takes a table name and a list of model equations inside curly brackets (if only one model equation is given, the brackets may be omitted). Some example model equations are shown in Listing 1. These models correspond to the H-D models (4a)–(4c) of Siipilehto (2000) and diameter increment models (3a)–(3b) of Nyyssönen and Mielikäinen (1978).

```
1 model tree { ❶
2     h = 1.3 + d^2 / (0.894 + 0.185*d)^2
          where s=1 ❷
3     h = 1.3 + d^3 / (1.811 + 0.308*d)^3
          where s=2
4     h = 1.3 + d^2 / (0.898 + 0.242*d)^2
          where s=3
5     d5 = (1 + 0.01*exp(5.4625 - 0.6675*
          log(stand.T) - 0.4758*log(stand.G
          ) + 0.1173*log(stand.D) - 0.9442*
          log(stand.Hdom) - 0.3631*log(d) +
           0.7762*log(h)))^5*d where s=1 ❸
6     d5 = (1 + 0.01*exp(6.9342 - 0.8808*
          log(stand.T) - 0.4982*log(stand.G
          ) +      0.4159*log(stand.D) -
          0.3865*log(stand.Hmed) - 0.6267*
          log(d) + 0.1286*log(h)))^5*d
          where s!=1
7 }
```

Listing 1: Height and diameter increment models in the DSL.

The first line (❶) specifies that the model equations inside curly brackets belong to the `tree` table, that is, they are tree-level models. Each line inside the curly brackets defines a model equation of the form (1). A model equation (❷) contains the output variable(s) on the left hand side and a mathematical expression on the right hand side. The `where` keyword specifies the condition under which the model can be applied. If the model can always be applied, the `where` keyword can be omitted. Variables in the same table can be referenced by the variable name, while variables in other tables by the table name and the variable name separated by a full stop. For example, on line (❸), the variables `d`, `h` and `s` are tree-level, while the other variables are stand-level.

Equations that are not simple formulas like those shown in Listing 1 must be written in another programming language and referenced using the `call` keyword. For example, the variables `Hdom` and `Hmed` refer to the mean height of the 100 largest trees and the height of the basal area median tree. The DSL lacks support for (naturally) expressing these computations. Listing 2 shows how they would be included via the `call` keyword if they were implemented as R functions.

The `call` keyword takes a language name, a function reference, and the function arguments. The syntax of the function reference is language-specific: the R language requires a file name and a function name, while native languages such as C and Fortran can take a either

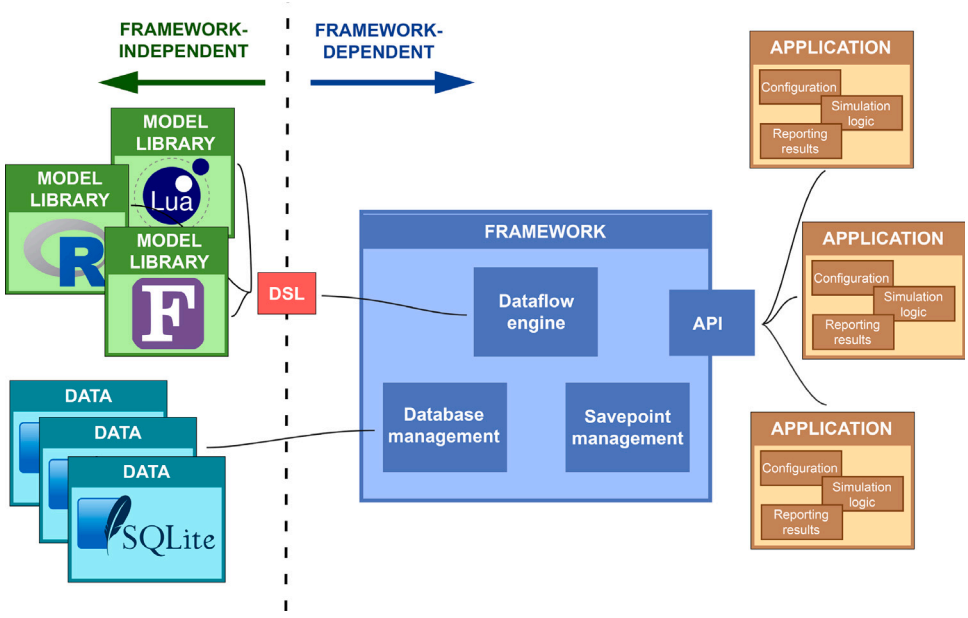


Fig. 2. An overview of the software architecture.

```r
Hdom<-function(d,h) {
  idx<-order(d)[1:min(
       length(d),100)]
  sum(d[idx]^2*h[idx])/
       sum(d[idx]^2)
}
```

```
model site {
  Hdom = call R["models.
       R":"Hdom"]
    (tree.d, tree.h)
}
```

Listing 2: Dominant height computation in R (left) and the corresponding DSL definition (right).

dynamic library file name and a symbol name, or a direct function pointer. The function arguments do not have to be variables: they can be any arbitrary expressions such as those shown in Listing 1. Out-parameters are supported using the `out` keyword. The model function must not have side-effects, that is, it must produce a value that only depends on its arguments.

3.3. Dataflow engine

The dataflow engine is implemented in Rust as an independent library that can also be used outside the framework. It parses DSL model definitions, builds and optimizes the dataflow graph, and compiles model chains to machine code for efficient execution (Fig. 3).

The DSL was specifically designed to allow aggressive optimizations: it lacks explicit control flow and side-effects, so the compiler has considerable freedom to reorder and delete code. While the DSL lacks explicit type annotations, the compiler infers data types for variables and expressions in order to produce efficient machine code. A variant of Algorithm W (Damas and Milner, 1982) with limited support for operator overloading is used for type inference. After type inference, the compiler lowers the graph into its intermediate representation (IR). The IR is based on the sea-of-nodes representation (Click and Paleczny, 1995) (an opcode listing is given in Appendix A.3). The optimizer performs function inlining, dead code elimination, common subexpression elimination and other simplifications on the IR. The DSL intentionally lacks explicit data structures: this allows the compiler to determine the memory layout only after the optimizer has eliminated redundant variables and models. Finally, the compiler computes an execution order using a variant of the global code motion algorithm described in Click (1995). Since the DSL has no side-effects, the compiler has much more freedom in choosing a good execution order than compilers for general-purpose programming languages. The Cranelift compiler backend (Bytecode Alliance, 2025) is used for machine code emission (including register allocation and instruction selection). We chose Cranelift over alternative backends (e.g. LLVM (Lattner and Adve, 2004)) due to its focus on compilation speed and security, as well as small binary footprint, although our implementation allows the backend to be easily changed, if deemed necessary in the future.

3.4. Lua API

The Lua API provides two high-level building blocks for applications: *transactions* and *savepoints*.

The first building block, transaction, is used for defining actions and observations. A transaction consists of SQL-like `SELECT`, `UPDATE`, `INSERT` and `DELETE` commands. A `SELECT` command observes values of variables, while `UPDATE`, `INSERT`, and `DELETE` modify the simulation state analogously to their SQL counterparts. Transactions are declarative: the application lists all actions a transaction may perform, but the simulator only executes those that affect the values of observed variables. Applications may augment the dataflow system (2) with additional rules to create generic transactions. For example, an application may define a rule such as "for all data variables x, if x_5 is a computed variable, then this transaction includes an action that replaces the value of x with x_5". This would allow the application to work with any model library that defines 5-year updates for some variables, regardless of which variables are updated.

The second building block, savepoint, is used for simulating multiple alternative branches from a certain simulation state. A savepoint is a sparse snapshot of the application's current state. At any time during simulation, an application can create a savepoint, and later restore the simulator state back to how it was when the savepoint was created. Savepoint creation is cheap: no data is copied at savepoint creation. Copies are lazily created when application state is modified after a savepoint has been created. The framework also provides a higher-level operator-based control API that applications can use instead of directly managing savepoints. Operators compose simple control flows into more complex ones. For example, the `all` operator combines control flows into a list of sequential actions, while the `any` operator branches on alternative control flows. Together these operators form a mini-language that can be used to describe many branching schemes. The composed control flow can then be passed to the framework for

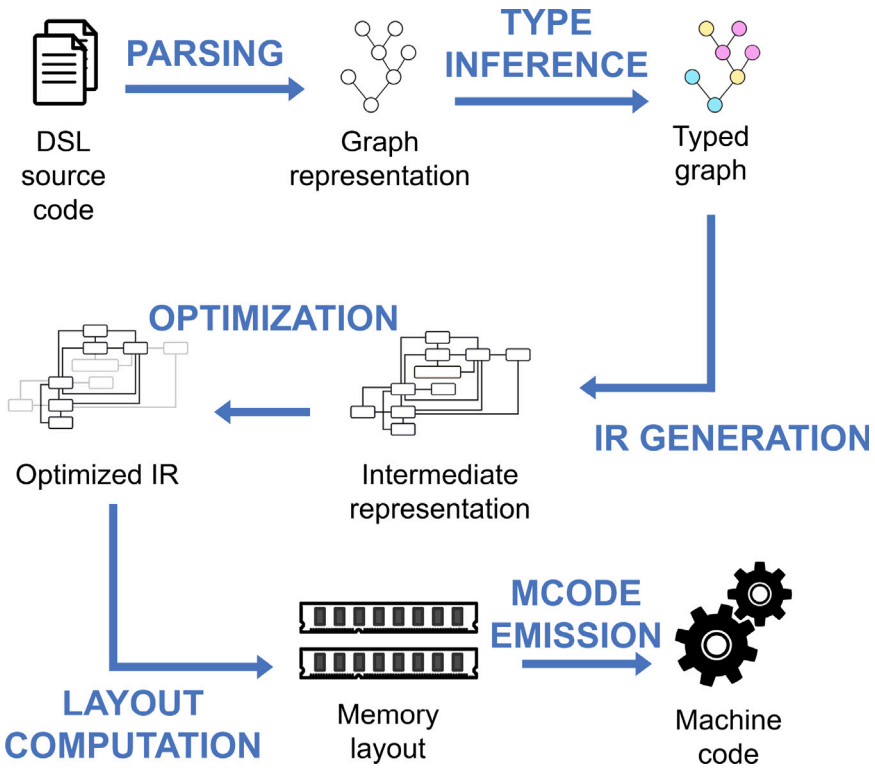


Fig. 3. The compiler pipeline of the dataflow engine.

execution, which internally interprets the control flow instructions and handles the required savepoint manipulation. Applications can also mix-and-match operator-based control flow and direct savepoint manipulation as needed.

4. Case study

We demonstrate the flexibility and computational efficiency of our framework via a sample application and three model libraries.

4.1. Simulator application

The application generates all feasible treatment schedules for a forest stand under user-configurable rules, and reports user-defined output variables. Users describe the set of feasible treatment schedules using a list of actions and conditions. A concrete example configuration is provided in Appendix A.4. The following actions are implemented:

- *Growth*: For each tree-level data variable `x`, if the model library defines a model for a variable named `grow'x`, replace the old value of `x` with the value of `grow'x`. If the model library defines any stand-level variable `ingrowth'y`, where `y` is either a `tree` or `stratum`-level variable, then create new trees or strata whose data variables receive the values of `ingrowth'y` for each tree- or stratum-level data variable `y`.
- *Thinning*: Replace the tree-level stem count `tree.f` with the result of a thinning model from the model library.
- *Clearcut*: Remove all trees.
- *Planting*: If the size of the `stratum` table is a data variable (that is, the data model contains strata), insert new strata. Otherwise, insert new trees. The initial values of data variables are given by silvicultural guidelines. Users can override the data level and initial values.

Users may request any combination of thinning, clearcut, and planting actions, while growth is called automatically by the application. The conditions for each action can be any arbitrary expressions that can be computed using available models. Users may also provide an early growth condition expression that controls the transition from `stratum` entities to `tree` entities. The application makes few assumptions about available variables, and only depends on the used models indirectly through naming conventions. The requested outputs can be any expressions that can be computed using available models.

Table 1
Primary model library (Model I).

Model	Reference
Growth (mature, mineral soils)	Based on Hynynen et al. (2014).
Growth (mature, peatlands)	Repola et al. (2018).
Mortality (mature)	Hynynen et al. (2002)
Early development	Based on Siipilehto et al. (2014).
Crown ratio, competition	Hynynen et al. (2014).
Biomass	Repola (2008) and Repola (2009).
Taper curves	Laasasenaho (1982).
Thinning models	Äijälä et al. (2014).

4.2. Model libraries

The primary model library (Table 1) contains models for natural processes, silvicultural guidelines, and auxiliary variables. Models are implemented in Lua, except for the early development models which are implemented in R. The model library comprises a majority of the code (5961 lines), compared to the application which is relatively small (330 lines, 5.2% of total lines of code). The model library is connected to the framework through 557 lines of DSL definitions. We have excluded lines that only contain whitespace or comments.

For technical evaluation of the software, we also implemented two additional growth model libraries (Table 2). We will refer to the growth models of Table 1, Pukkala et al. (2021), and Nyyssönen and Mielikäinen (1978) as Model I, II, and III, respectively. Model I consists of a relatively complex tree-level model chain for natural processes of

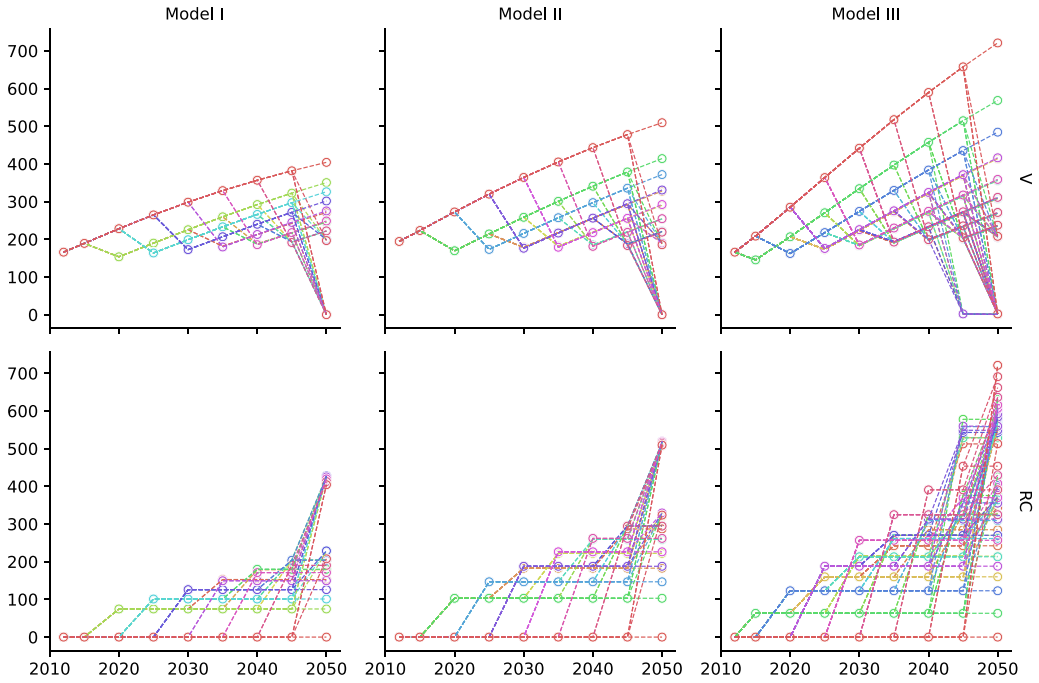


Fig. 4. Standing volume (upper row) and cumulative harvest volume (lower row) in m^3/ha for each model library. Each dot corresponds to a reporting year, and each line corresponds to a treatment schedule.

Table 2
Secondary growth models for technical evaluation (Model II & III).

Model	Reference
Growth, mortality, and ingrowth	Pukkala et al. (2021).
Growth	Nyyssönen and Mielikäinen (1978).

large trees, and a stratum-level model for early development. Model II is a growth model suitable also for uneven-aged forests, with tree-level diameter (but not height) growth, mortality, and ingrowth components. Model III has tree-level diameter and height growth without mortality or ingrowth. For Model II, we use the H-D model of Siipilehto (2000) to compute tree heights. Our Model II implementation is in Lua, and Model III in C.

4.3. Experiments

We perform a test simulation using each model library to verify that the framework is able to link it to the application. In each case, we swap the natural process models, but keep the auxiliary models (last four rows) from Table 1. For each model library, we record the data model and model chains determined by the model linking algorithm under different output requests and enabled treatments. We use measurement data from a stand located in Eastern Finland as input. However, we do not further analyze the model outputs, as our objective is to test the technical model linking algorithm, not evaluate the linked models or their compatibility. In general, we consider it the user's responsibility to ensure that the models they choose to use are compatible and properly validated.

We evaluate the computational efficiency of our framework through two comparisons against hand-written model implementations. We measure the CPU time and number of retired floating point arithmetic instructions (FLOPs) per growth step on a dataset of 20 stands. The purpose of measuring FLOPs is to be able to meaningfully quantify the amount of work performed by implementations using different programming languages and implementation techniques. The number of FLOPs represents the amount of computational work performed by each algorithm, while the CPU time includes all sources of overhead such as scripting language interpretation, cross-language function calls, and memory access patterns. All measurements are performed using the perf profiler on a Linux laptop with an Intel Core i5-1035G4 processor and 8 GB of RAM. We compare against implementations of models I and III. The former is a real-world production implementation written in Fortran and Pascal that has not been optimized for computational efficiency, while the latter is an ideal optimized C implementation created by us for this study. For model I, we turn off simulation of early growth and only use data with mature trees, because we do not implement the same version of the early growth model. We also turn off NFI calibration because it is not implemented in our version. For model III, the individual C model functions are shared, and the only difference between the two implementations is in the execution logic.

4.4. Results

The framework successfully links and runs the application with each model library, generating 34, 44, and 87 schedules with Models I, II, and III respectively (Fig. 4). Models with higher growth estimates generated more schedules because higher growth leads to more opportunities for treatments.

The complexity of the solved data model and model chains varied depending on the model library, requested output variables, and enabled treatments (Fig. 5). Without any treatments or output variables, the only maintained data variables were the current simulation year and time step length. In this case, the only work performed during simulation is incrementing the year: all simulation of natural processes is skipped, because nothing depends on their results. When treatments are enabled, the simulator must perform work even if no output variables are requested, because the number of branches depends on feasible treatments, which depend on variables updated by natural processes. When output variables are requested, additional variables not required by natural process must be maintained. For example, tree breast height age is used for biomass models, but not for growth in Models II and III.

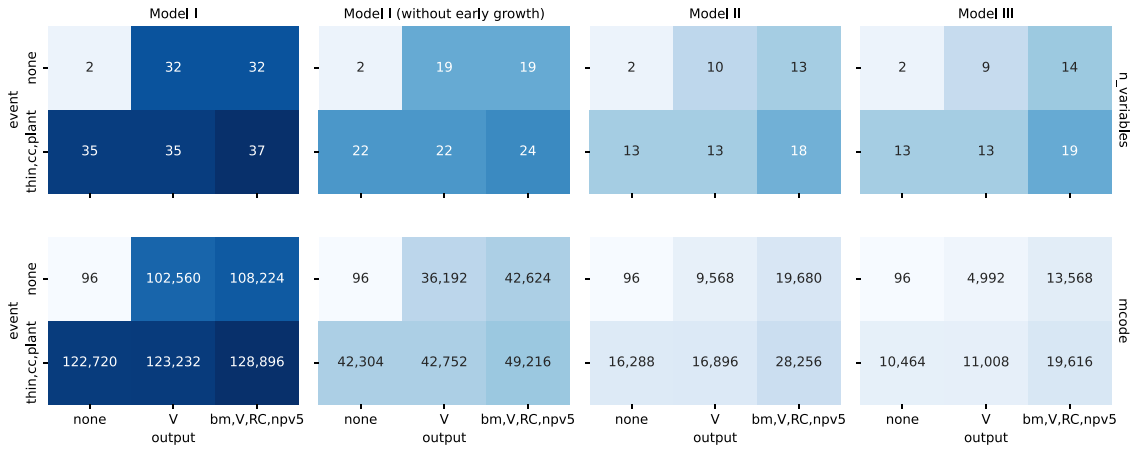


Fig. 5. Solved data model size (upper row, number of variables) and model chain code size (lower row, bytes of machine code) for each model library under different output variables (columns) and enabled treatments (rows).

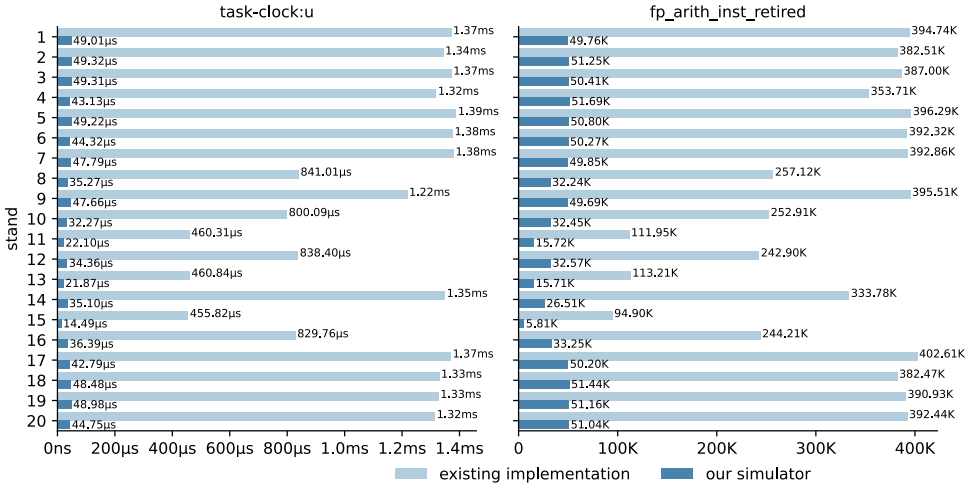


Fig. 6. CPU time (left) and FLOPs (right) per growth step for Model I. Average over 10,000 repeats of a sequence of 10 growth steps.

Our implementation of Model I was on average 27.8 times faster and required 7.9 times less FLOPs than the hand-written reference implementation (Fig. 6). The difference in FLOPs results from the elimination of redundant computations, while the CPU time also reflects low-level optimizations (e.g. memory layout) performed by the dataflow compiler.

Model III in our framework was on average 15% slower than our optimized C implementation (Fig. 7). Both implementation executed the same number of FLOPs, excluding very small differences caused by the Lua interpreter's use of floating point variables and a small number of redundant computations in the C version. The difference in CPU time reflects the overhead of the model chains generated by our dataflow compiler. Dividing the runtime difference by the number of model function calls per growth step, we get roughly 5 ns overhead per call, which translates to roughly 20 cycles at 3.7 GHz. The generated model chains perform additional bookkeeping about which values are currently computed, and dynamically load the model functions from a shared library at runtime, while the optimized C implementation was compiled together with the model functions, allowing the C compiler to inline the model functions. The Cranelift backend used by our dataflow compiler also performs less optimizations than the C compiler we used for the C code (GCC 15.1.1 with -O2).

5. Discussion

Our framework shows that a dataflow-based approach to automated model linking can enable the implementation of computationally efficient simulators while reusing existing model code and requiring less hard-coded logic and data structures than hand-written implementations. As a hand-written codebase changes over its life cycle, it accumulates complexity and inefficiency unless carefully maintained and refactored (Martin, 2017). In contrast, a model library is only a listing of models: there is no execution logic to maintain. Our comparison with the existing implementation (Fig. 6) shows the effect of such accumulated complexity on computational efficiency. The hand-written implementation used several times more FLOPs to compute the same results, because it computes values that are never used, and often the same values multiple times. While dynamic approaches often have considerable runtime overhead (e.g. Pereira et al., 2017), our comparison with the optimized implementation (Fig. 7) shows that JIT compilation of model chains can largely mitigate the impact. We hypothesize that the efficiency gap could further be reduced by switching to a more aggressively optimizing backend such as LLVM or libgccjit.

Augmenting the system of dataflow equations with application-specific rules (e.g. "the growth of a variable `x` is given by `grow'x`")

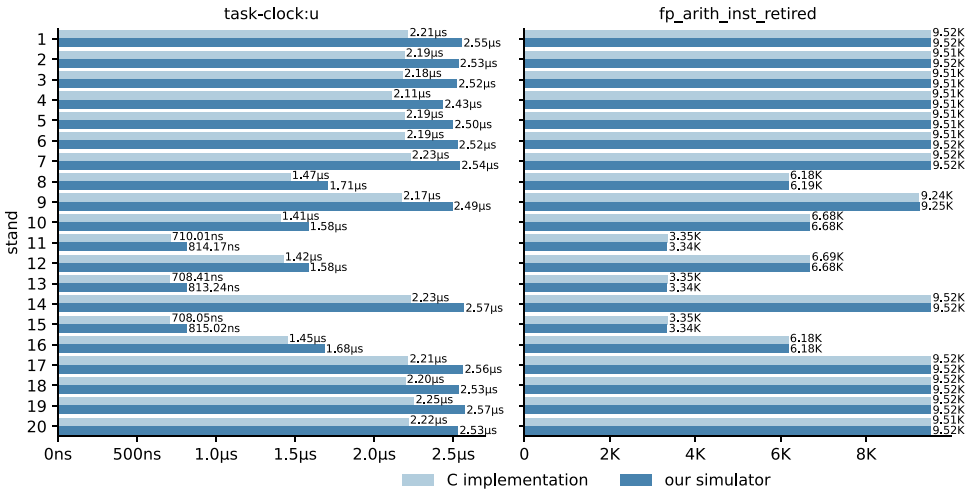


Fig. 7. CPU time (left) and FLOPs (right) per growth step for Model III. Average over 1,000,000 repeats of a sequence of 10 growth steps.

allowed the same application to work with the three model libraries we tested while making few assumptions about the data model. The complexity of the solved data model and model chains varied depending on the used model library, requested output variables, and enabled treatments (Fig. 5). A hand-written implementation accommodating the same use cases would either have to perform redundant work (updating variables that are not used), or increase complexity (checking which variables will be used). For example, some model libraries (e.g. Model I & III) include sub-models for both diameter and height growth, requiring height to be maintained as a data variable, while some (e.g. Model II) use a diameter increment model and a height-diameter model, allowing height to only be computed when required by an output variable. In general, it is also possible for a model library to include both types of models, making height simultaneously act as a data variable for some trees, and computed variable for some trees, which can be expressed in our framework by using an additional auxiliary computed variable (see details in Appendix A.1).

Many well-established frameworks, libraries, and platforms already exist for modeling and model linking (e.g. SIMO (Rasinmäki et al., 2009), Capsis (Dufour-Kowalski et al., 2011), OMS (David et al., 2013), OpenMI (Harpham et al., 2019; Moore and Tindall, 2005), EMF (Steinberg et al., 2008), k.LAB (Villa et al., 2017), SIMILE (Muetzelfeldt and Massheder, 2003)). The main difference between our framework and previous works is that the data model, dataflow, and control flow of model chains are all implicitly determined by the augmented dataflow Eq. (2), which are solved on a per-simulation basis. This allows the dataflow compiler to perform optimizations that would not be possible with an ahead-of-time compilation approach, and decreases the amount and complexity of code that has to be maintained. To our knowledge, implementing a JIT compiler to reduce the dispatch overhead of model function calls in model chains is also a novel approach.

The simplicity of our approach leads to several drawbacks. The main limitation is that models must be factored into individual pure functions. Existing code may require extensive refactoring to extract individual, independent model functions (e.g. Salminen et al., 2005). While there is evidence that pure functions have maintainability benefits (e.g. Dolado et al., 2003; Coblenz et al., 2016), they are also somewhat limiting in terms of expressiveness. Imperative constructs such as mutable state and loops require awkward workarounds. For example, consider an ingrowth function that keeps generating trees until some global condition becomes true, and each tree depends on the trees it has previously generated. Our DSL can neither express the loop or the internal state. The workaround is to place the loop in the application code, and the state in the model's input and output variables. This couples the application code to the model, and increases the complexity of both the application and the model. The DSL only supports numeric types to guarantee interoperability across multiple programming languages, which can be limiting for complex models. While this can be to some extent worked around with wrapper functions, wrappers both increase the amount of code that has to be maintained, and cause additional computational overhead, negating the benefits of our framework. In addition, we have no mechanism for validating the compatibility of linked models, making our framework prone to user errors such as models using the same variable with incompatible units.

A potential direction for future work is assessing whether imperative or stateful models could be naturally integrated in our dataflow framework. This could be approached by formalizing the workarounds currently required to implement them, similarly to the formalisms we presented in Section 2.2, and then integrating those formalisms into the dataflow analysis of Section 2.3. In addition, integrating semantic information (e.g. Villa et al., 2009) such as units and limits of variables could make the framework less error-prone for users by detecting incompatible models and providing semantic mediation e.g. via automatic unit conversions. Other potential future work involves exploiting the strengths of the dataflow paradigm. Dataflow introduces opportunities for parallelism, which was one of its original motivating factors (Johnston et al., 2004). Integrating automatic parallelization of model calls into the dataflow compiler could increase computational performance on large datasets, though such fine-grained parallelism also includes synchronization overhead, making it only beneficial when more coarse-grained parallelism (e.g. across stands) is not available. Dataflow is also well-suited for visualization, being the underlying paradigm for many visual programming environments. Building visualization tools or visual editors for our framework might increase its usability and accessibility to non-programmers.

6. Conclusion

We have introduced a framework for building forest simulator software using a dataflow-based approach. Unlike the traditional imperative and object-oriented approaches, we do not require explicit programming of model chains, or specification of an explicit data model. Instead, both are obtained as a solution to a system of dataflow equations based on a model library and use-case-specific output variables and actions. This makes the system flexible to changes, and reduces the amount of redundant computations, leading to high computational efficiency. The main limitation of our approach is that it requires models to be pure functions operating on numeric inputs and outputs, making it unsuitable for highly imperative models, or models operating on complex data structures.

CRediT authorship contribution statement

Tapio Lempinen: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Lauri Mehtätalo:** Writing – review & editing, Supervision, Methodology. **Annika Kangas:** Writing – review & editing, Supervision, Methodology. **Tero Heinonen:** Writing – review & editing, Supervision, Methodology. **Paulo Borges:** Writing – review & editing, Supervision, Methodology. **Jari Vauhkonen:** Writing – review & editing, Supervision, Methodology.

Software availability

1. fhk:
   - Name of software: fhk (***f**unktio**h**aku**k**one, function search engine)*
   - Developer: Tapio Lempinen
   - Contact: tapio.lempinen@luke.fi
   - Year first available: 2025
   - License: MIT License
   - Programming language: Rust
   - Source code: https://github.com/menu-hanke/fhk5
2. m3:
   - Name of software: m3
   - Developer: Tapio Lempinen
   - Contact: tapio.lempinen@luke.fi
   - Year first available: 2025
   - License: MIT License
   - Programming language: Lua, C
   - Source code: https://github.com/menu-hanke/m3

The model library and application described in Section 4 are available at https://github.com/menu-hanke/lim3 under the GNU AGPLv3 license.

Declaration of competing interest

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

Acknowledgments

T.L., T.H., and J.V. were financially supported by the Research Council of Finland Flagship UNITE (decision numbers 357906 and 359172). L.M., A.K., and P.B. were financially supported by the Research Council of Finland Flagship UNITE (decision numbers 357909 and 359174).

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.envsoft.2025.106661.

Data availability

Data will be made available on request.

References

Äijälä, Olli, Koistinen, Arto, Sved, Johnny, Vanhatalo, Kalle, Väisänen, Pentti, 2014. Metsänhoidon suositukset. Metsätalouden Kehitt. Tapion Julk. 179.

Antón-Fernández, Clara, Astrup, Rasmus, 2022. SiTree: A framework to implement single-tree simulators. SoftwareX (ISSN: 2352-7110) 18, 100925. http://dx.doi.org/10.1016/j.softx.2021.100925.

Bytecode Alliance, 2025. Cranelift. https://cranelift.dev/. (Accessed 24 March 2025).

Click, Cliff, 1995. Global code motion/global value numbering. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation. In: PLDI95, ACM, pp. 246–257. http://dx.doi.org/10.1145/207110.207154.

Click, Cliff, Paleczny, Michael, 1995. A simple graph-based intermediate representation. ACM SIGPLAN Not. (ISSN: 1558-1160) 30 (3), 35–49. http://dx.doi.org/10.1145/202530.202534.

Coblenz, Michael, Sunshine, Joshua, Aldrich, Jonathan, Myers, Brad, Weber, Sam, Shull, Forrest, 2016. Exploring language support for immutability. In: Proceedings of the 38th International Conference on Software Engineering. ICSE '16, ACM, pp. 736–747. http://dx.doi.org/10.1145/2884781.2884798.

Damas, Luis, Milner, Robin, 1982. Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '82. POPL '82, ACM Press, pp. 207–212. http://dx.doi.org/10.1145/582153.582176.

David, O., Ascough, J.C., Lloyd, W., Green, T.R., Rojas, K.W., Leavesley, G.H., Ahuja, L.R., 2013. A software engineering perspective on environmental modeling framework design: The object modeling system. Environ. Model. Softw. (ISSN: 1364-8152) 39, 201–213. http://dx.doi.org/10.1016/j.envsoft.2012.03.006.

de Kok, Jean-Luc, Engelen, Guy, Maes, Joachim, 2015. Reusability of model components for environmental simulation – Case studies for integrated coastal zone management. Environ. Model. Softw. (ISSN: 1364-8152) 68, 42–54. http://dx.doi.org/10.1016/j.envsoft.2015.02.001.

Dolado, J.J., Harman, M., Otero, M.C., Hu, L., 2003. An empirical investigation of the influence of a type of side effects on program comprehension. IEEE Trans. Softw. Eng. (ISSN: 0098-5589) 29 (7), 665–670. http://dx.doi.org/10.1109/tse.2003.1214329.

Donatelli, M., Rizzoli, A.E., 2008. A design for framework-independent model components of biophysical systems. In: Proc. Iemss 4th Biennial Meeting - Int. Congress on Environmental Modelling and Software: Integrating Sciences and Information Technology for Environmental Assessment and Decision Making, Iemss 2008. Vol. 2, pp. 727–734.

Dufour-Kowalski, Samuel, Courbaud, Benoît, Dreyfus, Philippe, Meredieu, Céline, de Coligny, François, 2011. Capsis: An open software framework and community for forest growth modelling. Ann. For. Sci. (ISSN: 1297-966X) 69 (2), 221–233. http://dx.doi.org/10.1007/s13595-011-0140-9.

Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John, 1994. Design Patterns. Addison Wesley, Boston, MA, ISBN: 978-0201633610.

Harpham, Q.K., Hughes, A., Moore, R.V., 2019. Introductory overview: The OpenMI 2.0 standard for integrating numerical models. Environ. Model. Softw. (ISSN: 1364-8152) 122, 104549. http://dx.doi.org/10.1016/j.envsoft.2019.104549.

Hasenauer, Hubert, 2006. Sustainable Forest Management: Growth Models for Europe. Springer, pp. 1–398. http://dx.doi.org/10.1007/3-540-31304-4.

He, Xiao, Wang, Xingwei, Shi, Jia, Liu, Yi, 2020. Testing high performance numerical simulation programs: experience, lessons learned, and open issues. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 502–515.

Hynynen, Jari, Ojansuu, R., Hökkä, Hannu, Siipilehto, Jouni, Salminen, Hannu, Haapala, P., 2002. Models for Predicting Stand Development in MELA System. In: Metsäntutkimuslaitoksen Tiedonantoja, (Number 835), Finnish Forest Research Institute, ISBN: 951-40-1815-X.

Hynynen, Jari, Salminen, Hannu, Ahtikoski, Anssi, Huuskonen, Saija, Ojansuu, Risto, Siipilehto, Jouni, Lehtonen, Mika, Rummukainen, Arto, Kojola, Soili, Eerikäinen, Kalle, 2014. Scenario analysis for the biomass supply potential and the future development of Finnish forest resources. In: Scenario Analysis for the Biomass Supply Potential and the Future Development of Finnish Forest Resources. In: Working Papers of the Finnish Forest Research Institute, (Number 302), Finnish Forest Research Institute, Vantaa, ISBN: 978-951-40-2487-0.

Johnston, Wesley M., Hanna, J. R. Paul, Millar, Richard J., 2004. Advances in dataflow programming languages. ACM Comput. Surv. (ISSN: 1557-7341) 36 (1), 1–34. http://dx.doi.org/10.1145/1013208.1013209.

Knapen, Rob, Janssen, Sander, Roosenschoon, Onno, Verweij, Peter, de Winter, Wim, Uiterwijk, Michel, Wien, Jan-Erik, 2013. Evaluating OpenMI as a model integration platform across disciplines. Environ. Model. Softw. (ISSN: 1364-8152) 39, 274–282. http://dx.doi.org/10.1016/j.envsoft.2012.06.011.

Laasasenaho, Jouko, 1982. Taper Curve and Volume Functions for Pine, Spruce and Birch. In: Communicationes Instituti Forestalis Fenniae, (Number 108), Metsäntutkimuslaitos, ISBN: 951-40-0589-9.

Lämås, Tomas, Sängstuvall, Lars, Öhman, Karin, Lundström, Johanna, Årevall, Jonatan, Holmström, Hampus, Nilsson, Linus, Nordström, Eva-Maria, Wikberg, Per-Erik, Wikström, Peder, Eggers, Jeannette, 2023. The multi-faceted Swedish heureka forest decision support system: Context, functionality, design, and 10 years experiences of its use. Front. For. Glob. Chang. (ISSN: 2624-893X) 6, http://dx.doi.org/10.3389/ffgc.2023.1163105.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, pp. 75–86. http://dx.doi.org/10.1109/cgo.2004.1281665.

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. Environ. Model. Softw. (ISSN: 1364-8152) 26 (10), 1240–1250. http://dx.doi.org/10.1016/j.envsoft.2011.03.011.

Martin, Robert C., 2017. Clean Architecture. Prentice Hall, Philadelphia, PA.

Moore, Roger V., Tindall, C. Isabella, 2005. An overview of the open modelling interface and environment (the OpenMI). Environ. Sci. Policy (ISSN: 1462-9011) 8 (3), 279–286. http://dx.doi.org/10.1016/j.envsci.2005.03.009.

Muetzelfeldt, R. I, Massheder, J., 2003. The simile visual modelling environment. Eur. J. Agron. 18, 345–358, URL http://www.simulistics.com/files/documents/SimilePaper.pdf.

Muys, Bart, Hynynen, Jari, Palahi, Marc, Lexer, Manfred J., Fabrika, Marek, Pretzsch, Hans, Gillet, François, Briceño, Elemer, Nabuurs, Gert-Jan, Kint, Vincent, 2010. Simulation tools for decision support to adaptive forest management in Europe. For. Syst. (ISSN: 2171-5068) 19, 86–99. http://dx.doi.org/10.5424/fs/201019s-9310.

Nobre, Silvana, Eriksson, Ljusk-Ola, Trubins, Renats, 2016. The use of decision support systems in forest management: Analysis of FORSYS country reports. Forests (ISSN: 1999-4907) 7 (3), 72. http://dx.doi.org/10.3390/f7030072.

Nuutinen, Tuula, Berger, Florian, Karjalainen, Antti, Lempinen, Reetta, Maltamo, Matti, Siitonen, Markku, 2010. Request-driven generation of calculation chains for adaptive forest analysis. Scand. J. For. Res. (ISSN: 1651-1891) 26 (1), 2–10. http://dx.doi.org/10.1080/02827581.2010.533691.

Nyyssönen, Aarne, Mielikäinen, Kari, 1978. Metsikön kasvun arviointi. Acta For. Fenn. (ISSN: 0001-5636) (163), http://dx.doi.org/10.14214/aff.7597.

Packalen, Tuula, Sallnaes, Petter, Sirkia, Seija, Kohonen, Kari, Salminen, Olli, Vidal, Claude, Robert, Nicolas, Colin, Antoine, Belouard, Thierry, Schadauer, Klemens, et al., 2014. The European Forestry Dynamics Model: Concept, design and results of first case studies.. Publications Office, LU, http://dx.doi.org/10.2788/153990, URL https://data.europa.eu/doi/10.2788/153990.

Pereira, Rui, Couto, Marco, Ribeiro, Francisco, Rua, Rui, Cunha, Jácome, Fernandes, João Paulo, Saraiva, João, 2017. Energy efficiency across programming languages: How do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. SPLASH '17, ACM, pp. 256–267. http://dx.doi.org/10.1145/3136014.3136031.

Pukkala, Timo, Vauhkonen, Jari, Korhonen, Kari T., Packalen, Tuula, 2021. Self-learning growth simulator for modelling forest stand dynamics in changing conditions. For. Int. J. For. Res. (ISSN: 1464-3626) 94 (3), 333–346. http://dx.doi.org/10.1093/forestry/cpab008.

Rasinmäki, Jussi, Mäkinen, Antti, Kalliovirta, Jouni, 2009. SIMO: An adaptable simulation framework for multiscale forest resource data. Comput. Electron. Agric. (ISSN: 0168-1699) 66 (1), 76–84. http://dx.doi.org/10.1016/j.compag.2008.12.007.

Repola, Jaakko, 2008. Biomass equations for birch in Finland. Silva Fenn. (ISSN: 2242-4075) 42 (4), http://dx.doi.org/10.14214/sf.236.

Repola, Jaakko, 2009. Biomass equations for scots pine and Norway spruce in Finland. Silva Fenn. (ISSN: 2242-4075) 43 (4), http://dx.doi.org/10.14214/sf.184.

Repola, Jaakko, Hökkä, Hannu, Salminen, Hannu, 2018. Models for diameter and height growth of scots pine, Norway spruce and pubescent birch in drained peatland sites in Finland. Silva Fenn. (ISSN: 2242-4075) 52 (5), http://dx.doi.org/10.14214/sf.10055.

Rizzoli, Andrea E., Donatelli, Marcello, Athanasiadis, Ioannis N., Villa, Ferdinando, Huber, David, 2008. Semantic links in integrated modelling frameworks. Math. Comput. Simulation (ISSN: 0378-4754) 78 (2–3), 412–423. http://dx.doi.org/10.1016/j.matcom.2008.01.017.

Salminen, Hannu, Lehtonen, Mika, Hynynen, Jari, 2005. Reusing legacy FORTRAN in the MOTTI growth and yield simulator. Comput. Electron. Agric. (ISSN: 0168-1699) 49 (1), 103–113. http://dx.doi.org/10.1016/j.compag.2005.02.005.

Segura, Marina, Ray, Duncan, Maroto, Concepción, 2014. Decision support systems for forest management: A comparative analysis and assessment. Comput. Electron. Agric. (ISSN: 0168-1699) 101, 55–67. http://dx.doi.org/10.1016/j.compag.2013.12.005.

Shifley, Stephen R., He, Hong S., Lischke, Heike, Wang, Wen J., Jin, Wenchi, Gustafson, Eric J., Thompson, Jonathan R., Thompson, Frank R., Dijak, William D., Yang, Jian, 2017. The past and future of modeling forest dynamics: From growth and yield curves to forest landscape models. Landsc. Ecol. (ISSN: 1572-9761) 32 (7), 1307–1325. http://dx.doi.org/10.1007/s10980-017-0540-9.

Siipilehto, Jouni, 2000. A comparison of two parameter prediction methods for stand structure in Finland. Silva Fenn. (ISSN: 2242-4075) 34 (4), http://dx.doi.org/10.14214/sf.617.

Siipilehto, Jouni, Valkonen, Sauli, Ojansuu, Risto, Hynynen, Jari, Miina, Jari, Saksa, Timo, 2014. Metsikön Varhaiskehityksen Kuvaus MOTTI-Ohjelmistossa. In: Working Papers of the Finnish Forest Research Institute, (Number 286), Finnish Forest Research Institute, ISBN: 978-951-40-2463-4.

Steinberg, Dave, Budinsky, Frank, Paternostro, Marcelo, Merks, Ed, 2008. EMF, second ed. In: The Eclipse Series, Addison-Wesley Educational, Boston, MA, ISBN: 978-0321331885.

Strîmbu, Victor, Eid, Tron, Gobakken, Terje, 2023. A stand level scenario model for the Norwegian forestry – a case study on forest management under climate change. Silva Fenn. (ISSN: 2242-4075) 57 (2), http://dx.doi.org/10.14214/sf.23019.

Villa, Ferdinando, 2007. A semantic framework and software design to enable the transparent integration, reorganization and discovery of natural systems knowledge. J. Intell. Inf. Syst. (ISSN: 1573-7675) 29 (1), 79–96. http://dx.doi.org/10.1007/s10844-006-0032-x.

Villa, Ferdinando, Athanasiadis, Ioannis N., Rizzoli, Andrea Emilio, 2009. Modelling with knowledge: A review of emerging semantic approaches to environmental modelling. Environ. Model. Softw. (ISSN: 1364-8152) 24 (5), 577–587. http://dx.doi.org/10.1016/j.envsoft.2008.09.009.

Villa, Ferdinando, Balbi, Stefano, Athanasiadis, Ioannis N., Caracciolo, Caterina, 2017. Semantics for interoperability of distributed data and models: Foundations for better-connected information. F1000Research (ISSN: 2046-1402) 6, 686. http://dx.doi.org/10.12688/f1000research.11638.1.

Wang, Wen J., He, Hong S., Fraser, Jacob S., Thompson, Frank R., Shifley, Stephen R., Spetich, Martin A., 2014. LANDIS PRO: A landscape model that predicts forest composition and structure changes at regional scales. Ecography (ISSN: 1600-0587) 37 (3), 225–229. http://dx.doi.org/10.1111/j.1600-0587.2013.00495.x.

Weiskittel, Aaron R., Hann, David W., Kershaw, John A., Vanclay, Jerry, 2011. Forest Growth and Yield Modeling. Wiley-Blackwell, Hoboken, NJ.