Comments on Designing a Language for Specifying Dynamic Models
Matthew N. White, December 5, 2023

1 Preface

This document is intended as an early draft of the design of a language for precisely representing models of intertemporal choice, and particularly to discuss with Pablo Winnant. The goal of the language is to be able to represent a wide array of intertemporal models in discrete time, using intuitive and clear syntax to represent mathematical ideas in a machine- and human-readable way. It draws on some of the style and syntax of `dolo` and `dolark`, which in turn inherit from `dynare`, but extends them to include more model description features. My intent is for the modeling format to look familiar and approachable to users of those packages, and to minimize the difficulty / complexity of reducing a model statement in our language into a `dolo`, `dolark`, or `dynare` statement through a parser.

A key philosophical difference between those packages and what we seek to accomplish here is that our model statements are *explicit* about implicit assumptions of (e.g.) `dolark`. This includes the recursive nature of a problem, as well as a statement of preferences (or more generally, how agent behavior is determined). The model structure explicitly states the problem being solved by an agent making a choice, rather than characterizing properties of their choice (e.g. arbitrage conditions).

The core design concept is a "block", representing a sequence of model steps or events that naturally go together in some sense. The sequentiality of a model is reflected by chaining blocks together with "connectors" that specify how the end of one block connects to the start of another, including control flow. Both `dolo` and `dolark` implicitly assume that there is only one block, and it necessarily connects with itself recursively. In this proposal, such an assumption must be made explicitly (but with minimal effort).

Like the predecessor model specification formats, our new language will use the `yaml` format to encode the model. Moreover, it will be recursive and modular in its design, so that model components (blocks and connectors) could be stored in the same file or different files and referenced by importation. Values and functions can be specified when a model block is created or filled in (or modified) later.

This draft only includes information about specifying a "micro model", with no description of how "macro objects" should be described (and the equilibrium values characterized). Such capabilities *are* an intended part of the modeling language (like in `dolark`), but are not reached here. The initial goal is to be confident that the widest array of microeconomic structures can be precisely specified in the language.

Finally, to maximize confusion, I shall hereafter refer to our new language as `pablo`.

2 Specifying a Block

Specifying a Block: When specifying a `pablo` block, it gets its own indentation level, often the top level. Just like a `dolo` model statement, a `pablo` block descriptively `explain` itself, but it must include a `label` by which the block can be referenced. There is no concept of time *within* a `pablo` block: any information from the "past" must be explicitly transmitted to it as a `state`.

Beyond those labels, the core sections of a `pablo` block are the `symbols` it uses, the `dynamics` governing what "happens" during the block, additional `definitions` that serve as algebraic macros, `functions` that are constructed as part of solving the model, details about how the `solution` should be solved numerically, and the `calibration` of parameters and other variables. These will each be addressed in turn.

2.1 Specifying Symbols

Just like `dolo`, the top of a `pablo` block provides a declaration of `symbols` that will be used within that block. Whereas `dolo` only asks for a declared list of each type of `symbols`, in `pablo` the modeler can (optionally) specify additional information about each symbol, including its domain and meaning, by using additional indentation levels. The categories of `symbols`, and the additional information that can be included for each, are as follows:

- `parameters`: Values that are constant *within* the block. If listed in the style of `dolo`, they are assumed to each be real-valued. If listed with an additional `yaml` indentation level, the modeler can specify a parameter's `domain` (e.g., $\mathbb{R}_+$ or $\mathbb{N}$) and `explain` its meaning. The `domain` can be used by model interpreter software to evaluate the validity of a calibration, and the `explain`ation can be included in documentation.

- `states`: Information that is "inbound" into the block. The information in these variables is presumed to *exist* by being drawn from the "past" (the prior block via a connector), but is not ever specified directly. If listed in the style of `dolo`, they are assumed to each be real-valued. If listed with an additional `yaml` indentation level, the modeler can specify a state's `domain` and `explain` its meaning, as with `parameters`.

- `controls`: Values that will be chosen by the agent. If the block is solved by a software pacakge, a mapping from the agent's information space (domain) to the control space (codomain) would be generated. As with the prior types, the modeler can optionally `explain` each control variables, as well as provide a `domain` and `codomain`. If this information is not provided, its domain is implied by the appropriate statement in `dynamics` (below), and its codomain is assumed to be $\mathbb{R}$.

- `functions`: Functional mappings that are created during the solution process, but are not themselves controls. This `symbols` category does not exist in `dolo` or `dolark`, instead being implicit as part of the solution method. If specified as a simple list, the (co)domain of each function should be inferred from the context in which it appears in the `dynamics`. Each of the `functions` can be optionally `explain`ed, and its `domain` and `codomain` can also be specified for clarity. Moreover, one or more of a function's first `derivatives` can be declared, naming other functions to specify the relationship among them.

- `distributions`: Distributions from which values that will be randomly drawn in the block. This is mostly a straight renaming of `exogenous` from `dolo`, to reduce confusion between the concepts of "exogenous shock process" vs "exogenous to the agent". These symbols can be defined as in `dolo` and `dolark`, or each `random` variable can be `explain`ed and have its `domain` (support) specified. Additionally, `pablo` differentiates between the *distribution* that is drawn on and the *variable* that holds a value from it.

- `next`: Objects (often functions) that are not defined *within* the block (and never could be) because they come from the *successor* block as part of the backward solution of the model (when appropriate). Objects named in `next` are referenced from the block's successor, potentially renamed through a `twist` (see below).

Note that other than the addition of `functions` and renaming `exogenous` to `distributions`, the only substantive change from `dolo` and `dolark` to `pablo` is the meaning of `states`. In `dolo`, the `states` are the inputs to the policy function– its domain. In `pablo`, they specify the variables that exist at the "start" of the block, akin to what were called "post-states" in HARK. If a block has (essentially) only a control action in it, then the `states` correspond to the usual concept of state variables.

2.2 Specifying Dynamics

In `dolo` and `dolark`, the `equations` section specifies both `arbitrage` equations that provide conditions on the controls that must hold, as well as the `transition` describing how the prior period connect to this one. The `arbitrage` conditions *implicitly* define the agent's problem by *characterizing the solution* to the problem; in `pablo`, the problem that the agent wants to solve to choose their control is always stated explicitly. The `transition` statement in `dolo` hardwires the idea that the model is recursive, connecting directly to itself in a loop. In `pablo`, we want to generalize the model's sequentiality, with the information from `transitions` handled in the `dynamics` section, while the concept of intertemporality or sequentiality is handled by a connector.

The syntax of the `dynamics` section is similar to that of `dolo` and `dolark` except that there is no reference to time `t`, which does not exist internal to a block. Each entry in `dynamics` should specify a single "step" of the model that is acted out in sequence. These "steps" can be one of the following:

- **Initialization**: The `start` directive indicates the moment of creation or model entry for the agent. A block that `start`s the model should not have any `states`, as nothing precedes it.

- **Explicit determination:** The value of a variable (on the LHS of an equation `=`) is explicitly determined by an algebraic statement on the RHS of that equation, using parameters, functions, and variables already determined in the block. The `states` are considered to exist at the outset of the block.

- **Implicit determination:** The value of N variables (named on the LHS of an assignment `:=`) is implicitly determined by the solution to an algebraic system of N equations specified on the RHS, in brackets and comma separated. Other than the variables implicitly determined by this system, the other symbols in each equation should already exist as parameters, functions, or variables already determined in the block.

- **Random determination:** The value of one or more variables (named on the LHS of a random draw $\sim$) is randomly determined by drawing from a `distribution` on the RHS. This is a very slight change from `dolo` and `dolark`, separating the distribution from the variables drawn from it.

- **Control choice:** The value of one or more control variables (named on the LHS of a function declaration `@`) is determined by the choice of the block's actor, based on variables named in parentheses immediately following the `@`– the agent's *information set* at the moment of choice, and consequently the domain of their policy function. The value of the control is implicitly determined (`:=`) by a problem on the RHS, often value maximization. Constraints on the choice can be specified with a vertical bar `|` after the problem, followed by a list of algebraic (in)equalities (comma separated, in brackets). As usual, other than the control variables in this step, the constraints should include only parameters, functions, and variables already determined in the block.

In general, the `dynamics` are an explicit statement of *what happens* in a block, step by step. It no more than a description of the sequence of events during the block, including a statement of the agent's "motivation" in making a choice of control.

2.3 Specifying Definitions, Functions, and Calibration

Both `dolo` and `dolark` allow the modeler to specify `definitions` that serve as algebraic "macros": mathematical objects that can be substituted into the `transitions` as warranted. This functionality should exist in `pablo` as well, with the sole change being the removal of time `t` references. In some cases, no additional value is added by including a `definition` rather than including it as a sequential step in the `dynamics`, but they can be useful when dealing with systems of equations.

Likewise, the `calibration` functionality in `pablo` should be identical to `dolo`, `dolark`, and `dynare`, with the addition that distributions can be specified in this block as well. That is, rather than specifying the nature of distributions in the `exogenous` block, the same information is instead passed as part of the `calibration`, as there is functionally no distinction between the two that was not already resolved by the `symbols` declaration. As in `dolo` (etc), additional variables can be defined by algebraic declaration.

Each entry in the `functions` block indicates a function declared in the `symbols` sub-entry to the left of a function declaration (`@`), followed by its domain variables in parentheses. The value of the function at any point in its domain is then determined (either explicitly `=` or implicitly `:=`) by an expression on the RHS.

2.4 Specifying a Numeric Solution Method

While the specification of control variables in `dynamics` describes how the agent makes decisions, it does not provide guidance on how the solution to the agent's problem should actually be *found* in practice. Directives of this kind can be provided in `pablo` in a block's `solution` entry, which contains (at minimum) a `method` entry. Additional entries might include choices for the `discretizations` and/or `approximations` used in the numeric solution method, or an initial `guess` of the `functions` to use as a starting point.

We leave all of this information intentionally vague here, as this is merely a draft proposal, not a functioning software package. The intended message is simply that information about how to *solve the model in practice* is designated separately from the statement of the*pure mathematical form* of the model.

2.5 Operations on Blocks

Once a block has been defined (declared), it does not constitute a model in and of itself. As originally declared, a block might not have all of its values "filled in"– it might only be a template without parameter values. For a lifecycle model, the modeler can create copies of blocks using YAML's notation for anchors and aliases. Parameter values can be individually filled in on these copied blocks using YAML's override feature. For example, in a lifecycle model, the same kind of block (period) might recur repeatedly, with the same mathematical structure, but some parameter values might vary by age.[1] Most importantly, a block can be `connect`ed sequentially to another block (or itself!) with a connector, discussed below.

3 Connecting Blocks

Connectors are the interstitial material between blocks– the interface between chunks of a model. This section describes how connectors work and how blocks can be assembled into a group by using connectors.

3.1 Connector Basics

Very little actually *happens* in a connector; the only operations or steps are as follows:

- **Relabeling:** Some variables determined in the predecessor block should be passed to the successor block's `states` via a simple mapping given by a `twist` dictionary. Any variable that was *not* renamed with twist is assumed to keep its same label (for the purpose of the successor block's `states` looking for it).

- **Time increment:** Einstein famously said that time is what prevents everything from happening all at once. If a connector has the `tick` declaration, this means that discrete

[1] We are developing a format for more parsimoniously specifying sequences of lifecycle parameters.

time advances during this connection. The absence of a `tick` declaration indicates that the next block takes place in the same segment of measurable time. A `tick` is used solely for accounting purposes, e.g. properly tracking simulated data or conveniently referencing aspects of the solution (in software that uses the `pablo` format as an input).

- **Flow control:** Sometimes an actor faces entirely different kinds of situations depending on events that are resolved as the model is acted out. Any model branching should be handled in a connector using the `?` operator. The `stop` directive can be used as part of flow control to indicate that the actor ceases operation.

The format for specifying a connector is a simple `yaml` statement. Its only fields are `twist` (a symbolic remapping) and an optional `tick` boolean (default `False`).

3.2 Assembling a Group

Blocks and connectors can be assembled using the `link` directive in a group environment–another top level `yaml` indentation. When assembling a model, a block or connector can be succeeded by a block, another connector, or nothing at all (in the special case of guaranteed termination in a connector).

Suppose that blocks named `block#` have been defined, as well as a connector called `connector`. Each of the following are examples of (potentially) valid syntax for the declarations in a `pablo` group that link blocks and connectors:

- `link(block0, connector, block1)` says that `block0` flows into `block1`; variables from the former are relabeled for the latter using the `twist` data in `connector`.

- `link(block0, block1)` says that `block0` flows into `block1`, and no symbolic relabeling is needed.

- `link(block0, connector, block0)` says that `block0` flows into itself recursively; the connector must appropriately label end-of-block values to the `states`.

- `link(block0, connector, d ? [block1, block2, block3])` says that `block0` flows into one of three blocks depending on the value of `d`, which should be 0, 1, or 2.

- `link(block0, connector)` says that `block0` will flow into *something*, but it is left undefined at this time.

- `link(connector, block1)` says that something will flow into `block1` using the relabeling from `connector`, but it is left undefined at this time.

A properly specified `pablo` group has up to one block (or connector) with an undefined predecessor and up to one connector with an undefined successor. That is, there is (at most) one way in and one way out– a defined beginning and end. If the group does not have a "way in", it should have a `start` directive in one of its blocks. A group is declared as its own YAML entry, listing the `link`ed blocks in its `content` entry.

3.3 Operations on Groups

Once a group has been declared, it acts much like a "superblock" and can be treated as if it were a block, in terms of its `states` (of its "entry block") and potential "outputs" (the variables determined within it). Groups can thus be linked together to form larger groups.

4 Specifying Agents

To specify a (population of *ex ante* homogeneous) agent(s) who make choices about their controls, the modeler does so by creating a YAML entry whose key data is the group referenced in its `model` entry. A well specified group to pass as a `model` for an agent type has exactly one `start` directive inside it, has no undefined successors nor predecessors, and is fully connected as a directional graph. That is, the model specifies where to start, can never "escape" its bounds (will `stop` or loop with certainty), and has no blocks that can't be reached.

An agent type might also specify the number or size of its population (e.g. that there are ten thousand such agents, or that there is a continuum of such agents with relative mass 3.6), as well as other data. These details are left unspecified as of this draft.

5 Example Model and YAML File

To make the draft specification more concrete, we include an extremely simple example model: an infinite horizon consumption-saving model with only transitory income risk.

5.1 Model Description

In each discrete period t, the agent receives stochastic labor income Y_t from distribution F, then chooses how much of their market resources M_t to consume C_t (yielding felicity via a CRRA function) and how much to save in assets A_t (with a risk free return factor of R), subject to a liquidity constraint. The agent seeks to maximize their expected present discounted value of time-separable utility, with discount factor β. Their problem is given by:

$$V(M_t) = \max_{C_t} U(C_t) + \beta \mathbb{E}[V(M_{t+1})] \ \text{ s.t.}$$

$$K_t = A_{t-1}, \qquad B_t = \mathsf{R}K_t, \qquad M_t = B_t + Y_t,$$

$$A_t = M_t - C_t, \qquad A_t \geq 0,$$

$$Y_t \sim F, \qquad U(C) = \frac{C^{1-\rho}}{1-\rho}.$$

5.2 Correspondence to PABLO

This basic model can be implemented in `pablo` with a single workhorse block, which contains all events for one period, and an initializer block that provides an entry point. The main block is recursively linked to itself with a simple connector, which relabels end-of-period A as beginning-of-period K and increments time. The model parameters are simply β, ρ, and R, and unemployment probability $\mho$; the only distribution is F (specified with a trivial discrete distribution for simplicity).

The `initializer_block` is trivial: it initializes a model agent with no wealth. The top of the `basic_CS_block` is largely self-explanatory and is intended to look a lot like `dolark`. The new features include an entry for how to discretely `approximate` the continuous domain of `kLvl` when appropriate, the link between the value function and the marginal value function, and the existence of a `next` entry that references objects from the block's successor (for solution purposes only).

As in `dolo` and `dolark`, the `definitions` entry provides algebraic substitutions that can be used when solving or simulating the model. Note that I have created variables for the utility flow, marginal utility, and continuation payoff as part of the `definitions`. The `dynamics` break the sequence of the model into small bits– probably smaller than you might expect. In the model, bank balances `bNrm` are the interest factor times capital holdings, income `yLvl` is drawn from the simple shock distribution, and market resources `mLvl` are the sum of bank balances and income flow. The choice of `cLvl` is specified as spending on `mLvl` and solves the value maximization problem subject to the liquidity constraint.

In the `functions` entry, we define the beginning-of-block (marginal) value function over K_t. Note that this does not correspond to the ordinary Bellman value function $V(M_t)$ in the model statement. It is the expected (marginal) value of *entering* the period with K_t in capital (previous A_{t-1}), taking account of the income risk that is about to be resolved. The typical Bellman value function could be constructed if the workhorse block were split into smaller blocks, but this document provides the simplest implementation.

The `solution` entry provides information on how the model should be solved. In this case, it indicates that the endogenous grid method (EGM) should be used and includes a discretization for the end-of-period state variable `aLvl`, as well as a trivial guess as a starting point. The `calibration` entry just below the `solution` should have no surprises.

The crux of `pablo`'s model structure is that blocks are linked to each other in a sequence, which might recur on itself. For this very basic model, we need only one connector, labeled as `time_loop`, that connects the consumption-saving block back on itself. The `twist` entry says that `kLvl` in the successor block should be mapped to `aLvl` in the predecessor block, and `vFunc_next` in the predecessor block should map to `vFunc_now` in the successor block (for backward solution purposes). The only required links are to chain the initializer into the main block (with no connector needed), and the main block back on itself (through the `time_loop` connector). This specifies the microeconomic model, which can then be given to a consumer type with 10,000 agents.