Comments on Designing a Language for Specifying Dynamic Models
Matthew N. White, October 4, 2023

1 Preface

This document is intended as a very preliminary first draft of the design of a language for precisely representing models of intertemporal choice. 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. To the greatest extent possible, I have tried to adhere to the style and syntax of `dolo` and `dolark`, which in turn inherit from `dynare`. My intent is for the modeling format to look as familiar as possible 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.

A key philosophical difference between those packages and what I seek to accomplish here is that I want to make our model statements *explicit* with respect to the 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). Moreover, there should be a clear separation between statements about dynamics vs the characterization of behavior.

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 our modeling language, 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.

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, usually the top level. Just like a `dolo` model statement, a `pablo` block can include a descrip-

tive `name`, 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, the `felicity` attained during it, 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 (for some specification of how the agent makes a choice), 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 not chosen by the agent. These can be left symbolic in the model statement, and then filled in later as part of the calibration.

This `symbols` category does not exist in `dolo` or `dolark`. 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.

- `distributions`: Distributions from which values that will be randomly drawn in the block. This is 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.

Note that other than the addition of `functions` (which can be ignored by being left empty) 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 I called "post-states" in HARK.

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 determination of controls (a policy function or decision rule) is specified at a higher level. The `transition` statement 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 within `transitions` handled in the `dynamics` section, while the concept of intertemporality is handled by a connector.

The syntax of the `dynamics` section is identical 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. 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 an action statement `!`) is determined by the choice of the block's actor, based on variables named in parentheses immediately following the `!`. Constraints on the choice can be specified with a vertical bar `|` 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, without any specification of how the control choice is made.

2.3 Specifying Definitions 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.

2.4 Specifying Preferences

For the purposes of this draft, I propose that the specification of within-block preferences be identical to the optional declaration in `dolo` and `dolark`, using the `felicity` field to specify a utility flow.

2.5 Operations on Blocks

Once a block has been defined (declared), it is not immutable, and it does not constitute a model in and of itself. For a lifecycle model, the modeler can `copy` a block one or more times to make identical replicates. Additional values or distributions can be specified by using the `calibrate` command on it, passing the additional mapping. 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.
- **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 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 for intertemporal discounting, properly tracking simulated data, and 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 `remap` (or `twist`?), a symbolic mapping, 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.
- `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.

3.3 Operations on Groups

Once a group has been declared, the modeler can `copy` it and modify it (or its copies) with the `calibrate` directive. Any parameter values passed in this way are recursively assigned to the blocks (or groups) inside it, if they have `symbols` that match those passed. Moreover, a group 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 the group). Groups can thus be linked together to form larger groups.

The primary use case for copying and linking groups is to form a lifecycle model in which the structure of each period is identical, but (some of) its data varies over time. While I am not specifying it in this document, I am confident that we can devise a convenient format for specifying time-varying parameters and have them easily imported into a finite horizon sequence of groups.

4 Specifying Agents

To specify a (type of *ex ante* homogeneous) agent who makes choices about their controls, the modeler does so by creating an `agent` in `yaml` format. The key entry is the group referenced in the `model` field. A well specified group to pass as a `model` for an `agent` 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.

To specify *how* the agent chooses values for its controls, additional information must be provided to close the agent's model. I am still thinking about how to design a more general system of describing intertemporal preferences and beliefs. For the moment, let's stick with just additively separable utility with time-consistent discounting, plus rational expectations. Hence the only field that needs to be defined is an entry for `discount`, which should be either a single real number or an appropriately long list of reals.