Institute for Food and Resource Economics

University of Bonn

Discussion Paper 2025:3

Reflections and recommendations on Software Quality Management in economic models

Wolfgang Britz[a], *Torbjoern Jansson*[b], *David Schäfer*[c]

[a] *Institute for Food and Resource Economics, Bonn University, Germany.*

[b] *AgriFood Economics Centre and Department of Economics, Swedish University of Agricultural Sciences, Uppsala, Sweden*

[c] *EuroCARE Bonn GmbH, Bonn, Germany*

Editor: Thomas Heckelei

Institute for Food and Resource Economics

University of Bonn

Nußallee 21

53115 Bonn, Germany

Phone: +49-228-732332

Fax: +49-228-734693

E-mail: thomas.heckelei@ilr.uni-bonn.de

Reflections and recommendations on Software Quality Management in economic models

Wolfgang Britz[a], *Torbjoern Jansson*[b], *David Schäfer*[c]

[a]*Institute for Food and Resource Economics, Bonn University, Germany.*
[b]*AgriFood Economics Centre and Department of Economics, Swedish University of Agricultural Sciences, Uppsala, Sweden*
[c]*EuroCARE Bonn GmbH, Bonn, Germany*

Abstract

Software Quality Management (SQM) for economic simulation models includes different aspects of software development, such as using a version control system, developing, and applying coding guidelines, proper documentation of code, and a testing strategy to systematically locate and remove errors. This paper focuses on testing as the main element of SQM to decrease the probability of incorrect outcomes and reduce debugging costs. Other elements of SQM are discussed more briefly and in relation to testing. Due to differences in developer competence, in software used for coding, and in the organisational structure of research projects, approaches for testing in industry projects are not always applicable to economic models. Based on real-world examples, we show how a testing strategy can be implemented in economic modelling, spanning from automated checks and tests to a release strategy for the economic model. Overall, we recommend developing a test strategy for an economic model as early as possible to save cost and to ensure reliable model outcomes.

Keywords: Software Quality Management, Testing, Economic Modelling
JEL classification: C8, C88

1 Introduction

The correctness and replicability of results generated with scientific tools is vital for building trust in policy impact assessments (Podhora et al. 2013). In the field of economic research this trust is questioned by the replication crisis, which criticizes incentive structures and research conventions that prevent

replication studies and fosters practices such as selective result reporting and using designs with low statistical power (Ferraro and Shukla 2020, Kasy 2021, Finger et al. 2023, Storm et al. 2024). While that discussion revolves primarily around empirical methods, economic simulation models face similar challenges in maintaining credibility. Many journals require researchers to provide the data and code used in their economic empirical and simulation studies for replication purposes (Christensen and Miguel 2018, Journal of Global Economic Analysis 2024). However, the code size of economic simulation models often makes it difficult to assess their quality and the correctness of the generated results. Hence, to ensure the accuracy and reliability of the results, they should adhere to specific research software quality standards. These standards are supposed to enable researchers to understand, replicate, and further improve upon existing and new research (Anzt et al. 2021). The aim of this paper is to introduce approaches and tools to facilitate the development of high-quality research software in economic simulation models, drawing on concepts of software quality management (SQM) and change management. This addresses a gap in the literature as little has been written about good scientific practices in the development and application of economic models, even if their results can have significant impacts on policy debates, such as the article by Searchinger et al. (2008) on land use impact of biofuel mandates.

Researchers face strong dependencies on own developed and third-party research software (Hannay et al. 2009). Faults in such software are not a mere nuisance but can impact published results. Soergel (2015) gives an example of an assumed software with just 1,000 lines of code in which one line comprises an error. This is well below the reported error rates in complex software projects (Boehm and Basili, 2007). He assumes further that this error has a 10% chance of meaningfully changing the outcome for any input data set (i.e., not letting the program finish with an error) and that a researcher will consider, on average, 50% of the wrong outcomes as plausible. This results in a 5% chance of wrong output. He argues further that error propagation in software code is likely, such that results are likely "inaccurate, not merely imprecise" (Soergel et al. 2015). This gives a further argument for a more rigorous approach to SQM for economic simulation models with their broad societal impact.

Economic simulation models contain software code and can be subjected to the same quality management considerations as other software products. In the literature, related considerations include, for example, the definition of relevant software quality criteria such as accuracy, reproducibility, reliability, and usability (O'Regan 2019, Antz et al. 2021). Further, they entail quality control measures with key components of testing and code reviews and other quality management aspects such as versioning and release strategies in change management (O'Regan, 2019).

This paper focuses on using quality control, primarily testing, to ensure reliable research and discusses concrete approaches for testing. Our focus on testing reflects that searching for and correcting errors is a large part of software development and maintenance costs. These costs tend to be higher the

later errors are found in the development cycle (Westland 2002). For instance, if incorrect model results are detected only when a report is almost finished, many working steps need to be repeated and result sections re-drafted.

The paper is structured as follows. First, we introduce the specific role of SQM for economic simulation models compared to industry software development. Second, we present key SQM elements relevant to economic simulation models as research software, including a thorough description of test types and test strategies. Finally, we discuss the different implementation strategies and the general role of economic simulation models in the domain of research software and arrive at some recommendations. Concrete examples of testing strategies for the code of two economic simulation models are presented in annexes: continuous testing in FarmDyn, a single-farm bio-economic model, and testing of stable releases in CAPRI, a partial equilibrium model.

2 Economic models as research software

2.1 *Comparison with industry software*

Although basic concepts and challenges are similar, the development of an economic simulation model, from here on referred to as economic model, differs from that of general software in some crucial ways, as we illustrate in Table 1, with consequences for how SQM is handled. Table 1 summarises some differences, focusing on those posing challenges to SQM in economic modelling, acknowledging that professional software developers partly face the same problems (O'Regan, 2019).

The developers have different *aims* and *competences*. In the software industry, the deployed code is often the main output of the department, if not the company. An internal or external client defines requirements for newly developed or adapted code. Developers have some training in computation or software engineering (e.g., Exter and Ashby, 2019), which lets them use established approaches to prototyping, code development, testing and deployment, etc. Their project managers have incentives to consider aspects such as future maintainability and reusability of code parts in overall code design. SQM focuses on ensuring that the client's expectations in the software are met, defined by the client's requirements.

For most researchers, including those working with economic models, the software is merely a tool to create their main outputs, such as scientific articles. Clients, such as research project sponsors, will hardly scrutinise this code. Instead, the perceived quality of an economic model foremost depends on a methodologically sound approach, correct and up-to-date data, and an empirically derived proper parameterisation. The typical researcher working on economic models has no formal training in software engineering, develops code for her research and might not expect to work with the same model afterwards. Research project leaders have mostly no training in software engineering, either, but senior

expertise in fields such as methodology or research project management. This might lead them to delegating central decisions about software design and implementation to (junior) researchers, such as which language or packages to use or overall code design. Initiatives to improve the quality of the model's code are mostly indirectly honoured, for instance, by allowing for more refined results or avoiding delays from using erroneous or inefficient code. Depending on researchers' competence, incentives, and management, considerable differences emerge across economic models concerning coding style or aspects of SQM.

Another key aspect impacting SQM is differences in programming languages used. Most industry software projects use general-purpose programming languages, such as C++, C#, or Java, which require formal training for proper code design and efficient coding. Their code is structured in callable functional units that operate with each other by interfaces that define inputs and outputs. This structure is supported by encapsulation and scoping. Such language features facilitate the development of libraries which can be shared across projects and the construction of test suits, which are collections of pairs of inputs and expected outputs of a functional unit. Tools related to SQM, such as for testing or automated documentation of the code structure, complement the languages, along with tutorials on how to use them.

In contrast, economic models here are often developed in highly specialised Algebraic Modelling Languages (AML) such as GAMS[1] or GEMPACK[2]. They allow efficient definition/declaration of equations in economic models and related data manipulations without lengthy training or deeper knowledge of software engineering, which makes them popular tools. Overall, code design in an AML is often a minor issue, at least as long as the project remains small, and the model is not modular. AMLs largely lack encapsulation concepts to facilitate unit testing (Britz and Kallrath 2012). Tasks such as data preparation and reporting for the core model are hence not realised as functions or similar but instead carried out by a sequence of compiled or interpreted statements that manipulate variables with global scope. The core of an economic model consists of a system of equations and/or inequalities that are simultaneously solved or act as constraints to an optimisation. It is often hard or impossible to test these equations independently. Few tools related to SQM are available for AMLs, if any, and related information is scarce. However, error-prone constructs such as pointers are absent in AMLs, and a syntax close to mathematical notation eases writing self-explaining code. These differences blur if developers embed code of general-purpose programming languages into their AML scripts, as is possible in GAMS with Python, for example.

[1] https://www.gams.com/

[2] https://www.copsmodels.com/gempack.htm

Table 1: Comparison of software development for an economic model with an industry software product

		Economic model	Industry software
Aim of software development		Provide simulation results feeding into academic theses/papers or policy briefs.	Meet software requirements with functionality and usability as agreed with internal or external clients.
Competence profiles	Primary developer	Trained economist, typically with no or little knowledge of software engineering	Software engineer
	Project lead	Senior economist with typically limited knowledge of software engineering	Senior software engineer
Focus	Primary developer	Contributions to own career such as papers under the guidance of project lead (methods, data), coding to produce results for current project	Deliver code adhering to aims and standards set by the project lead (including SQM)
	Project lead	On-time delivery of project deliverables such as reports, ensure state-of-the-art methodology and up-to-date data use	Meeting project timelines and agreed software requirements, overall code design, ensure future code maintainability
Programming software	Type	Algebraic modelling languages (GAMS, GEMPACK, AMPL…) requiring limited training	General-purpose programming languages (C++, Java, Python…) requiring intensive training
	SQM	Software structure makes unit testing difficult; no or limited SQM tools and related tutorials are available.	Software structure supports unit-to-system testing, ready-to-use SQM tools as part of language, related (online) tutorial.
Development cycle	Drivers	New research project or application	New requirements
	Link to software development	Often not clear if code from the current project will be used in future ones	Building on existing code (maintainability) is essential to be cost-effective

Source: The authors

As all symbols have global scope and the concept of functions or similar is missing, GAMS and GEMPACK largely prevent publicly shared libraries or packages such as those found for Python or R. Instead, communities of economic modellers have gathered around specific implementations of model

types, such as in the case of global CGE models around GTAP, GLOBE or MIRAGE. Each such model has its specific pros and cons when analysing a research question but combining components from different models to create a new model is difficult. A key reason is that the building blocks of each model, such as the code for simulation equations, their benchmarking[5] and post-model reporting cannot be easily ported across models as these blocks are not encapsulated. This has triggered a discussion around more generic modelling platforms, which require, for instance, modularity to support different methodological solutions for model parts and/or to add extensions on demand (Britz and Van der Mensbrugghe 2018, Britz et al. 2021b) or to deploy portable modules for specific trade model specifications to be used in equilibrium models (Britz et al. 2021a).

2.2 *The development cycle of economic models*

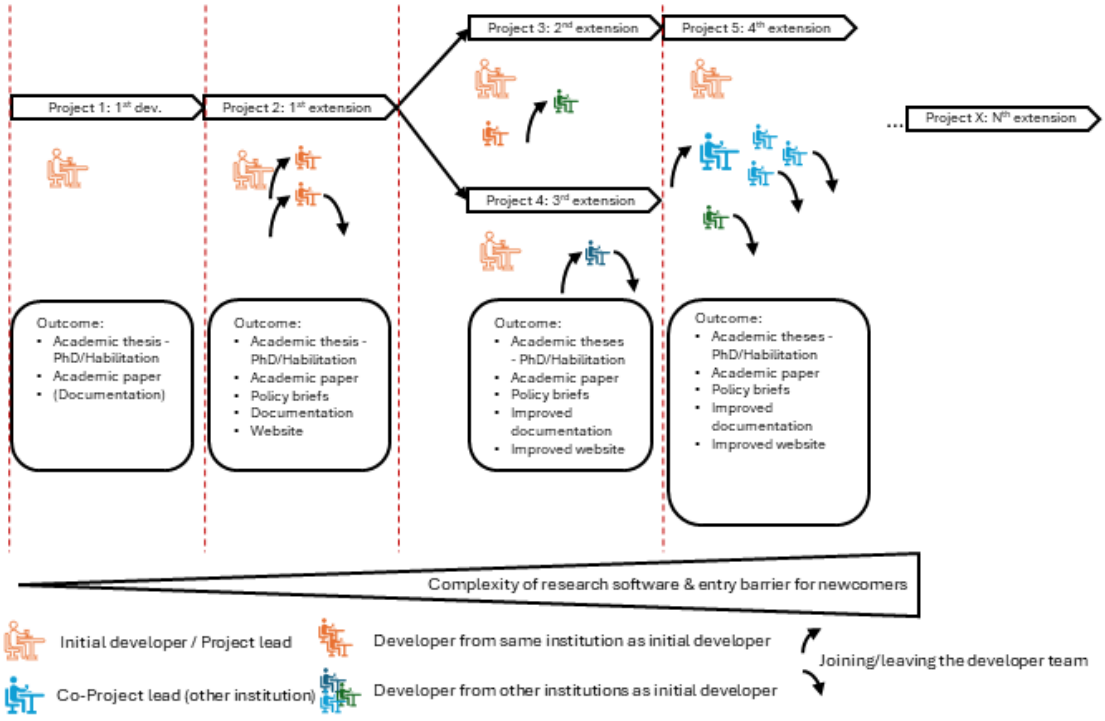
Industry software and economic models are often developed in project cycles, with development periods following one another. However, rooted in diverging objectives, competencies, and foci, as discussed above, the economic model development cycle exhibits specific characteristics that delay or even inhibit the initiative of a sophisticated SQM or test strategy in the early to mid-development phase.

Figure 1 illustrates, in a stylised manner, our understanding of the project cycle for an economic model. A similar pattern is described in the discussion by Baxter et al. (2012) on the "Research software engineer." Many economic models evolved originally from a PhD thesis in 3 to 5 years, with a PhD student writing the model's code, procuring necessary data, and estimating parameters to generate results. Contingent on the student's ambitions and success, a second project might extend the economic model and apply it to new topics, involving new researchers with little to no programming experience. As for the initial developer, their fundamental goal is further academic qualification. Some researchers leave the development team after one project, while others stay and continue using the model to take further steps in their academic careers. Once the team and number of projects reach a critical size, falling fixed costs allow for new activities, such as dedicated websites on the model. Combined with an increasing number of academic papers, this lets the model develop its brand and attract new projects and contributors, potentially working at different institutions. As the number of developers across institutions and projects grows and with them the overall code size of the model, deficits of missing software quality management become apparent, such as hard-to-read and to-debug code due to different personal coding styles and missing documentation, diverging model versions with incompatible

[5] In economic modelling, a benchmark means an assumed equilibrium situation to which the model is calibrated so that it becomes the optimal solution if no shock is introduced. Benchmarking is the process of creating the benchmark, for instance by calibrating behavioural equations or adjusting balances and flows.

features, and a growing number of errors. At the same time, newcomers face increasing barriers to successfully work with the model's code.

Figure 1: Development-/Life cycle of an economic model



Source: The authors

3 Key SQM elements for economic models

Quality in software comes from quality in the processes that develop it (O'Regan 2019). There are several models for software processes and lifecycles, such as the Waterfall model and Agile. Here, we describe a selection of tools and concepts that, in our experience, are important for producing quality software for an economic model, regardless of the development lifecycle model.

3.1 *Tests*

Unit tests are fundamental for testing in the industry (Runeson 2006). They feed predefined inputs into functional units and compare outputs to known results. Defining these control data lets coders consider the range of potential inputs, related error conditions, and outputs early, which leads to the concept of test-driven development (Beck 2002). The example below, taken from the Junit package for automated unit testing in Java (Hunt and Thomas 2003), demonstrates a test for a code unit:

```
@Test
void testCalculatorAdd(){assertEquals(2,calculator.add(1,1));}
```

The test function called "*testCalculatorAdd*" checks if using the pair of arguments "1,1" on the method "add" of the class named "calculator" returns the results "2". Unit tests also check how code handles non-regular input, for instance, special values such as INF. In the following exposition, we will frequently use the term *check*. By a check, we mean code in the production software that throws an error if certain conditions for input or output data are not satisfied. Tests are often run to see if such checks are triggered.

The example with *testCalculatorAdd* above underlines the challenges of applying unit testing to the equation system of an economic model. It assumes that the results returned by the model (or a smaller block of model equations) for a given set of input data and a shock are known for certain, which is hardly the case when code development starts. However, test cases with expected outputs exist, such as a benchmark check for which no infeasibilities should occur or a no-shock check that should replicate the benchmark. Unit testing can also be applied regularly to code for transformations of input data or benchmarking with their clearly defined tasks, such as producing closed market balances. Besides conceptual challenges related to unit testing; the grammar of GAMS also hinders their use. Whereas functional units in general-purpose language have clearly defined interfaces based, for instance, on arguments passed into functions, this concept does not exist in the grammar of GAMS. This implies that other approaches to testing must be used in AMLs; we propose checks built in the code besides testing whole applications.

The International Software Testing Qualifications Board (ISTQB, 2024) lists seven principles of testing that we find worthwhile to cite here.

- *Testing shows the presence of mistakes*. Testing aims to detect defects within software but can never remove all defects. However, it can reduce the number of unfound issues.
- *Exhaustive testing is impossible*. It is impossible to test all combinations of data inputs, scenarios, and preconditions within an application.
- *Early testing*. The cost of an error grows exponentially throughout the stages of the Software Development Lifecycle, motivating testing as early as possible.
- *Defect clustering*. Errors are typically clustered in certain modules; often, around 20% of the modules comprise 80% of the errors. Modules where errors are found should, therefore, be tested extra carefully.
- *Pesticide paradox*. After errors detected by a test are fixed, rerunning it cannot help to find new issues. Testing strategies need to be reviewed and updated regularly.
- *Testing is context dependent*. It must focus on the key aspects of quality management, with reliability essential for research software.
- *The absence-of-errors fallacy*. Error-free software is not necessarily successful—there is a difference between doing things right (no errors) and doing the right things.

3.1.1 What to test.

A testing strategy for an economic model requires a precise definition of the model's code base, which can prove challenging due to project-specific files used to prepare data or represent shocks. The fraction of the code base covered by a set of tests is a metric called "test coverage" or "test completeness", and it is desirable to have a high-test coverage. Tests should cover the whole production chain, which prepares input data and parameters, solves the model, and carries out reporting. Equally, it must consider all options which add or remove code on demand, such as in the case of different reporting options or modular model extensions[6]. A test strategy must reflect that code changes at any point of the chain can affect outcomes in follow-up code. Projects need additional tests for code not covered by the central strategy. Assuming existing code is properly tested, testing becomes necessary once code is added or changed and might require updates to the tests.

3.1.2 Test strategy and types of checks and tests

Different types of tests or checks address different classes of errors. *Compile-time errors* are caused by syntactically incorrect code and cannot affect outcomes as they prevent code execution. Related checks are fast as they don't require code execution. Detected by the compiler, these errors are mainly dealt with during code writing. Conditional use of whole files or code passages hides code from the compiler such that all potential configurations need to be tested. *Run-time errors* let the executing program throw an error. Again, they cannot result in wrong outcomes but might severely delay a research project. Run-time tests are more computing-time intensive as actual model runs are needed. Both types of errors are technical; the software typically indicates offending statements, which eases their correction. This is typically not the case for the third type of error, which we term *outcome errors*. In contrast to runtime errors, checks for them cannot be part of the production code. They relate to results considered incorrect given the input data. Simulation runs with an economic model produce typically large results sets such that outcome errors might go unnoticed even if implausible.

The test strategy is jointly defined by designing tests that address the different types of errors, deciding what events in the development cycle should trigger tests, and assigning responsibilities to run tests and deal with detected errors. Related efforts must be balanced with competing measures to improve the quality of research. Designing a test strategy must also consider the required resources for

[6] For CGEBox, a modular platform for CGE modelling, the number of GAMS files used in a simulation run with the same shock files and database varies between 35 and 100, depending on which modules and reporting options are present. At the same time, the number of code lines (not considering comments) increases from around 12.000 to 30.000. In parallel, due to conditional use of code passages, different sections of the same file might be in use.

its maintenance, such as human capital, hardware, and software. The following paragraphs will detail the different test types and related test strategies.

3.1.3 Compile-time checks and tests

Compile-time checks rely on syntax checks built into the compiler and are addressed mainly during code development. Code compiled during project specific developments often comprises only part of the whole code base. Due to the global scope of all symbols in AMLs, changes in symbol names, their dimensions, or the sets defining their domains during development can provoke compile-time errors in other parts. This motivates again a testing strategy that encompasses all possible code configurations.

AMLs allow the coder to program compile-time tests, which are useful to raise errors that otherwise provoke run-time errors, to save time for the user, and to ease understanding the cause of potential errors. For instance, testing that required files exist at compile time, which are later used at run time, can avoid unnecessary computing time[7]:

```
$$if not exist "required.gdx" $abort "Necessary file "required.gdx" not found,
file: %system.fn%, line: %system.incline%"
```

It can also be helpful to refrain from further compilation after compile-time errors occurred to avoid a flood of follow-up errors by inserting a line as follows at the end of each file:

```
$$if not errorfree $abort "Compilation error after file: %system.fn%"
```

Compile-time tests are easy to assess as they fail (some error or warning raised) or not. They are fast as they do not execute the code. A testing strategy for an economic model should, therefore, cover all possible model configurations, as illustrated by the FarmDyn example in Annex A1.

3.1.4 Run-time checks and tests

Like compile-time checks, run-time checks are partly built into the software itself, such as mathematical error trapping, and complemented by checks coded by the developer. Tests with predefined test sets ensure that coded checks work correctly and that all cases are caught, which otherwise throw run-time errors by the software. Coders tend to catch cases that otherwise would result in run-time errors with if clauses (the $ operator in the GAMS code below), such as:

```
yield(crop) $ land(crop) = production(crop)/land(crop)
```

This example of a check is deliberately poorly designed. Both negative land inputs and crop production without land use go unnoticed and might provoke follow-up errors. A better solution addressing these

[7] Examples are coded in GAMS, but the underlying ideas can be ported to other AMLs as well.

issues is shown below. The computation of yield, highlighted, in grey is only a minor part of the code; it mainly consists of checks[8]:

```
Loop(crop,
    curCrop(crop) = yes;
    if(production(crop) gt 0,
         if(land(crop) le 0,
              abort “Production, but land use <=0 ”,curCrop,land,production,
                   "file: %system.fn%, line: %system.incline%, ";
         else
              yield(crop) = production(crop)/land(crop);
         );
    elseif production(crop) lt 0,
         abort “Negative production ”,curCrop,production,
           "file: %system.fn%, line: %system.incline%";
    elseif land(crop) <> 0:
         abort “No production, but non-zero land use “,curCrop,land,
           "file: %system.fn%, line: %system.incline%";
     );
);
```

Besides mathematical error trapping, run-time checks by the software itself might relate to input/output errors, such as missing files or full disks. Errors and warnings by AMLs can also relate to model generation and solving. Catching warnings related to solves and solver statistics, such as on the number and size of infeasibilities, avoids otherwise unnoticed error propagation and useless follow-up computations, or worse, analysts reporting results of unsuccessful model runs. Not catching unsuccessful solves should be considered faulty code. The following code snippet shows an example where the coder has decided to accept model solutions with a maximal infeasibility below a certain threshold and otherwise throw an error:

```
abort $ (myModel.maxInfes > 1.E-4) "Infeasibilies ",myModel.maxInfes,
      "file: %system.fn%, line: %system.incline%";
```

To avoid errors, for instance, related to benchmarking and solving the model, developers might want (and need) to add a plethora of additional run-time checks, such as testing for missing input data as shown in an example below (GAMS code), which first checks if there is any region without capital demand (symbol *p_cap*) and then populates a dynamic set *rs* to report the regions where the problem occurs:

```
if(sum(reg $ (not p_cap(reg)),1),
    rs(reg) = yes $ (not p_cap(reg));
    abort "Missing capital demand for regions (rs) in %dataset%.gdx",rs,
```

[8] A singleton set *curCrop* is used to report the offending crop and a display of the set *crop* in the loop will show the full list.

```
            "file: %system.fn%, line: %system.incline%, ";
);
```

Similar checks can test for non-negativity or mutual compatibility between different data items (exhaustion conditions such as closed market balancing, value = price times quantity, etc.).

The examples above checked for regularity of inputs. Compile-time checks can also be added after a code sequence to assert that it has worked as intended, i.e., to test for bugs in the code. The following piece of GAMS code, for instance, can be executed after code balancing a SAM[9] to assert that the SAM is indeed balanced against a chosen accuracy threshold *acceptedError*, reports the offending cases and stops further code execution (*is* and *js* are the accounts of the SAM):

```
parameter sambal(is),colSum(is),rowSum(is);
colsum(is) = sum(js,sam(is,js));
rowsum(is) = sum(js,sam(js,is));
sambal(is) = colSum(is) - rowSum(is);
colsum(is) $ (abs(sambal(is)) < acceptedError) = 0;
rowsum(is) $ (abs(sambal(is)) < acceptedError) = 0;
sambal(is) $ (abs(sambal(is)) < acceptedError) = 0;

abort $ card(samBal) "SAM not balanced", samBal,colSum,rowSum,
                          "file: %system.fn%, line: %system.incline%";
```

Catching cases of inputs or results of data transformations with undesired properties as early as possible in the code is recommended to avoid time-consuming debugging exercises of resulting follow-up errors. Run-time checks should also address required properties of final model outcomes, such as for closed market balances or non-negative prices. Reports that, for instance, generate a SAM from the results of a Computable General Equilibrium (CGE) model should throw an error if the resulting SAM is not balanced. This can prevent faulty outcome from being analysed; reported imbalances also can help to find conceptual errors in model equations or post-model reporting. Test-driven development requires to conceptualise these checks before code development, to define desired or undesired testable attributes of inputs and outcomes of (new) code, as highlighted by examples above.

Run-time checks can test for successful model calibration by generating – but not solving – the model at benchmark values and throwing an error in case of infeasibilities. For CGE models, a Walras test as part of the equation system can check for correct economy-wide exhaustion, and a run with a numeraire shock can test for homogeneity. It is recommended to develop coding guidelines that help to

[9] Social Accounting Matrix: A way to report all flows in an economy between accounts such as production sectors, household(s), government, and the Rest of the World, with revenues of each account in the rows and related expenditures in the columns. According to the principle of National Accounting, each account must be closed, a state termed a balanced SAM.

design and implement run-time checks; the examples underline that these recommendations are partly specific to certain model types.

3.1.5 Outcome tests

Run-time checks should include automatically detectible errors in outputs produced, such as negative prices or violated exhaustion conditions. In contrast, what we call an *outcome test* typically requires a human to decide on the plausibility of model results. Exemptions are code changes not intended to affect model outcomes, such as code updates aiming at faster execution or improved code clarity. They can be tested in an automated manner for no change in results, which is an example of a unit test. However, most code changes in economic models aim to improve specific model results or add new ones. Some changes against previous results are, therefore, intended, but other results can be affected in unforeseen ways. Whether results are improved and not worsened might depend on model configuration, parameterisation, or input data. This holds especially true for models comprising integer variables that show a highly non-linear, jumpy response to changes in model structure and parameterisation. Those models are np-hard to solve (Fischetti and Luzii 2009) such that the time needed to find a (quasi-) optimal solution for a specific model instance is hard to judge beforehand, challenging the design of a test strategy.

3.1.6 Stability tests

Stability tests, as a specific class of outcome tests, assess if outcomes differ under different hardware and software, to ensure that results depend deterministically on model structure, data, and parameters, only. Potential causes for such instabilities are, for instance, models that are not globally convex or comprise flat sections in constraints or the objective function such that solutions differ across solver versions. Programming blunders can also produce instabilities, such as accepting non-optimal or infeasible solutions. Constrained optimisation models might be solved repeatedly with bounds changed programmatically after time-outs. These changes in the solution space can imply that the final solution depends on the number of trials, which in turn are driven by the soft- and hardware used or current processor load. Faulty code might assign start values from old solutions on disk to variable instances no longer comprised in the current model which then later are erroneously reported as part of its solution. To address such issues, stability tests run the same process, for instance, on different hardware and software versions, and collect and compare the resulting solutions.

3.1.7 Performance tests

Performance tests (Vokolos and Weyuker, 1998) reflect that performance is one aspect of model code quality, such as the time needed to benchmark and solve the model and the required memory and disk

space. Performance matters for at least four reasons. First, the importance of sensitivity analyses is increasing, which requires solving the model many times. Second, if the model is too slow to solve, a solver might not find a feasible or optimal solution. Third, model developments require repeated runs with changes in structure or parameterisation, and long solution times might limit possible improvements in a given time. Finally, users might react to long run times by reducing model size and complexity. Besides less detailed results, this typically implies increased aggregation bias (Britz and Van der Mensbrugghe 2016). The chosen hardware and software tend to be more relevant for performance compared to stability testing. Stability testing informs about minimum requirements for secure replication, and performance testing about necessary requirements to achieve desired performance levels.

3.1.8 Resources needed for testing

Setting up the test strategy is a one-time activity, and updating the test instances is necessary only when new projects start, which require their test instances or after larger structural changes. That leaves regular testing as the major workload. Running tests and checking for failed ones is a repeated activity with more or less known time requirements, but the resources required to address failed tests are largely unknown beforehand. Compile-time and run-time errors are easily detected and often corrected quickly. Correcting implausible results found by outcome tests can be more challenging, especially where changes to model structure or parameterisation are required. Failed outcome tests can also detect combinations of inputs and/or model configurations that cannot be handled, and new checks can be added to the code to prevent that the model is executed on them.

Independent of the type of test, fixing errors can provoke new errors elsewhere. Equally, existing errors in follow-up code might be detected only after preceding errors are removed, and more code is executed. It is therefore recommended to repeat tests with all follow-up programs after fixing errors in the production chain. Moreover, tests should log results in a report indicating what was tested and the outcome. That way, test metrics such as test coverage can be developed over time and regression tracked (O'Regan, 2019).

3.2 *Software Version control*

Software version control (SVC, Zolkifli et al., 2018) ranked highest as a topic in evaluating good practice in software development in life sciences (Artaza et al. 2016). It is based on tools that store the history of individual files in a software project and make it accessible from a central repository via an internet protocol. This allows for the synchronisation of files in a project across machines and coders. If a user fetches the content from the central repository, only updated files are downloaded and integrated into their so-called local working copy. The SVC software will automatically replace newer versions of

files from the central repository if their local copies do not comprise any changes. Thanks to semantic parsing, the tools can often successfully merge changes from the server in specific parts of a file into a local version comprising modifications in other parts. If this fails, the user must deal with the resulting conflicts. So-called commits of a file or group of files from a local machine to the central repository increase the internal version number and store besides the changes to the files' content, a timestamp, the user ID, and a log entry. Suppose errors later can be linked to a particular commit. In that case, the log informs about who committed the code, which is essential to generate incentives for testing and to assign, where possible and appropriate, responsibility for error correction.

SVC software also supports so-called *branches*, which are separate streams of code development in one software project. Branches are essential, for instance, to allow for the development of new features outside of the main code and to maintain stable releases, as discussed below. SVC allows rolling back all or some files to a specific commit in a local working copy. If version control is consequently used, this allows to later get all files back used to produce results for a project or paper without the necessity to freeze a copy of the whole code. While commits are usually automatically labelled with some internal identifier, a user can *tag* them with a topical label, for instance, to indicate major or minor versions of the code. So-called *commit hooks* allow starting test runs automatically when code is uploaded. This is possible even before new code is added to the central repository, so commits can be rejected if tests fail. Combined, these functionalities render an SVC an essential part of SQM. Several SVC protocols exist, such as GIT[10] and Subversion[11] (SVN).

3.3 *Release strategies*

By a *stable release* of a model, we mean a version that (i) never changes and (ii) is made available to others other than the developer. A release strategy outlines how and when releases are produced. Bug fixes trigger a new minor or maintenance release. Typically, a new release comprises a "sufficient" number of changes to the model and is deployed only after passing a predefined set of tests, where "sufficient" can be defined based on subjective or objective criteria. Having version control in place facilitates the development of a release strategy because it allows one to identify each revision of the code and to work with branches of parallel development. Releases are also connected with certain types of testing.

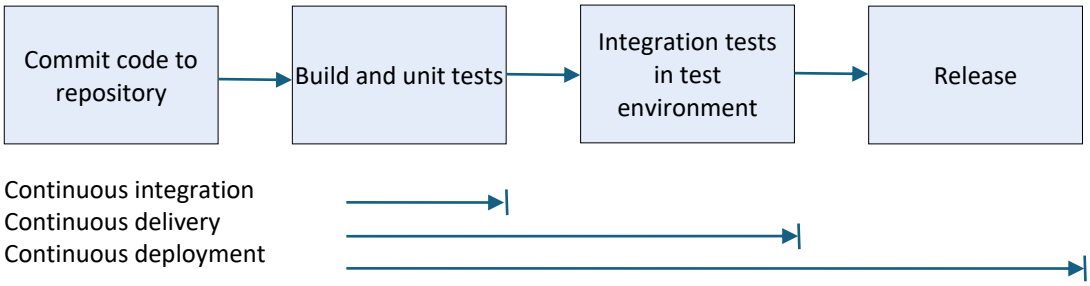
In commercial software development, it is increasingly common to automate the release strategy so that specific tests are automatically carried out when the code base has been sufficiently revised to

[10] https://git-scm.com/

[11] https://subversion.apache.org/

motivate a new release. Depending on the scope of the strategy, it is common to distinguish three levels of automation, illustrated in Figure 2. *Continuous integration* is a strategy for continuously integrating new developments into a central code repository, combined with automated build (compilation) and unit tests. *Continuous delivery* takes the automation one step further by deploying the integrated builds to a more evolved test environment where the entire system is tested, which is called integration testing. If approved by the release manager, the code can then be deployed. We speak about continuous deployment if that final approval is also automated.

Figure 2: Different scopes for automated testing and release



Source: The authors

SVC is widely used in economic modelling. As it is difficult to develop unit test suites in an AML, as discussed above, automated unit testing for each code change is less common. Automated integration tests of the entire software for each commit are unlikely if the computing time for running through all steps amounts to many hours. Therefore, the automated testing required for continuous integration, and at any rate for continuous delivery, is typically missing. Nevertheless, suppose the production version of the model is derived from the central repository. In that case, it is not uncommon for committed code to enter the production version immediately without systematic testing or release, leading to fuzzy versioning where it is unclear which version of the model and data was used for a particular application.

As discussed next, a good release strategy can prove valuable for a network of researchers developing a model over many years for several reasons. Many of those reasons are also valid for an individual researcher developing their model code in the context of some study.

First, a release strategy supports stability and predictability: The shared code base is in constant flux when many researchers collaborate on a modelling project. A mature model accumulates many features over time, and not all features are always functional or tested in the head revision of the SVC trunk. When a researcher starts working on an applied analysis with such a model, she needs to decide which version of the model is the most suitable for the study at hand. The answer is not always the head revision of the SVC, which might contain bugs in parts not recently used or tested or changes to raw data that cause model results to change. Suppose the model has a stable release that has been thoroughly

tested upon publication and not changed. In that case, such a release is a good starting point for an applied study that does not critically depend on the latest features or data.

Second, a release strategy supports peer review: If studies are carried out with a released model version, reviewers and readers can access it from a central repository, easing compliance with requirements to publish data and publication methods.

Third, a release strategy supports the network's viability: New users can be directed towards a tried-and-tested model version with known properties. Stable releases are also useful for training courses, as exercise results don’t change.

Finally, a release strategy can reduce testing costs by not testing everything for every commit. Instead, an extensive set of tests is carried out with each new release only. In that way, it is possible to test batches of commits that introduce new features, fix bugs or other unforeseen consequences of the new features, or data updates to several areas.

3.4 *Documentation*

Documentation serves *internal* aims related to code development and maintenance, often called technical documentation, and *external* ones to inform software users and others. We distinguish two types or scopes of internal documentation. The first type is comments within the source code. While code must be programmed using the grammar and vocabulary of the software language, it should be written as far as possible in a “self-documenting” style (see section 3.5 on coding guidelines). So-called in-line comments provide complementary information in natural human language to inform more granularly what specific statements or code passages do from a conceptual viewpoint and motivate the specific code implementation (Khamis et al. 2013). Comments should not simply repeat verbally what the follow-up statements do. They are important during debugging and reviewing code as they speed up its understanding. They can also help to find conceptual and other errors and to structure the code visually. These viewpoints provide an obvious link to SQM.

A second type of internal documentation specifies relations between code elements (functions, objects, packages, their organisation in files and folders, etc.) not (directly) visible from the code itself and beyond single files. For modern software languages, this type of documentation is typically produced automatically by utilities, which are part of the language itself. Such utilities generate additional documents beyond the code, often strongly hyperlinked. They refer mainly to the architecture

of the software and less to its granular implementation[12]. Internal documentation can contain technical terms or abstract concepts to inform developers and not users, such as diagrams coded in Unified Modelling Language or database diagrams.

External documentation to users bridges the conceptual layers of software functionality and code implementation. For research software, this external documentation is often called methodological documentation and typically comprises model equations in mathematical notations, including their derivations (Herrmann and Fehr, 2022). A methodological documentation should not require that readers know the software language in which the model is coded. However, such knowledge is needed to check for coherent code implementation given the documentation. Moreover, external documentation includes instructions on using the model code, including installation instructions. It should also cover how input data needed for the model must be organised concerning format, content, and location in the file system, and it should comprise information on how to use a graphical user interface or text-based approaches to steer the model code.

Different teams organise the model documentation differently. For instance, methodological documentation and a user manual can go hand-in-hand to inform how parameters belonging to blocks of equations are handled in external files, etc., or can be separated. There is also a growing tendency to use Wiki-based documentation instead of providing one document. Wikis make it easier to provide specific documentation for certain model releases. Still, depending on the granularity of the Wiki pages, they are harder to read than a classical document. It is sometimes possible to compile one document from the Wiki pages. Important functionalities of good documentation, such as hyperlinking different sections and topics and providing a searchable index, can be found in both approaches.

3.5 *Coding guidelines*

Coding guidelines specify rules on writing software to ensure a commonly agreed upon and thus less personalised programming style, including naming symbols, structuring code in functional units, commenting, and organising code in files and folders (Long et al. 2013, Green and Ledgard 2011). Such guidelines are especially relevant in research projects where coders have no formal training in software engineering such that they are unaware of accepted rules for writing code. Guidelines, hence, aim to produce code that can be more easily understood and maintained. They can also comprise hints to

[12] The documentation tools as part of GGIG (Gams Graphical Interface Generator, Britz 2014) provide part of this functionality for GAMS projects, generating a set of strongly hyperlinked web pages. The tool model2tex from gams.com (https://www.gams.com/48/docs/T_MODEL2TEX.html?search=model2tex) can generate LaTeX code of a model's equation, including substituting the name of GAMS symbols by names more commonly used in mathematical notation, such as Greek symbols.

improve performance, such as speeding up execution or reducing memory load. Due to the specific language features of AMLs, guidelines from general-purpose software languages need to be adjusted to become sensible. Multiple examples of coding guidelines for GAMS exist[13], spanning from single internet pages with some valuable hints to documents with dozens of pages and quite detailed rules. Comparing them reveals that teams and authors sometimes have differing views on what constitutes a desired coding style. This might reflect the specific nature of the projects the guidelines are intended for, such as the size of their overall code base, model type, degree of modularity, and personal preferences. Developing and applying such guidelines is an integral part of SQM for an economic model but discussing them in detail is beyond the scope of the paper.

3.6 *Integration across projects*

Costs of reconciliation of code developed in multiple projects are hard to allocate. Moreover, re-integrating project-specific code contributions (beyond bug fixes) in a "master version" often hardly benefits current projects but is instead an investment into future ones (Pidd 2002). It makes the most sense when combined with regular efforts to update data and parameters. Such continuous maintenance and development have worked well in number of cases, such as for the global Computable General Equilibrium (CGE) Model GTAP (Hertel and Tsigas 1997) and its variants, for the single country CGE IFPRI standard model (Löfgren et al. 2002), and to some degree for CAPRI (Britz and Witzke 2014), a partial global equilibrium model for agri-food markets which is regionalised to smaller administrative units for Europe. The FarmDyn model, used as an example in the annex 1 for a testing strategy, might be on a similar path (Britz et al. 2021a). The mentioned examples are template models where the same set of equations can be applied to many different data sets. Developing and maintaining such a template model allows distributing the fix costs of model development and maintenance over many research projects (Britz et al. 2021b). Repeated applications help to generate a trademark for the model, for instance, by peer-reviewed publications on which follow-up projects or papers can be built. Realising that the model (code) is an asset increases incentives to develop good documentation and handbooks, offer courses, etc., and build a user community around an economic model.

[13] Batmodel project: https://www.batmodel.eu/d7-2-common-guidelines-for-documentation-and-coding/
CAPRI: https://capri-model.org/ts/dokuwiki/lib/exe/fetch.php?media=red_book_on_gams_style.pdf
GAMS.com: https://www.gams.com/latest/docs/UG_GoodPractices.html
Christope Gouel: https://github.com/christophe-gouel/gams-style-guide
Paul Natsuo Kishimoto: https://github.com/mit-jp/style-guides

Long-term maintenance of the same model, including its software code, raises additional challenges. After years of development, the code in use has mainly been written by people no longer part of the current development team. Newer code might draw on software features that were previously unavailable. This can provoke, besides personal preferences, differences in coding style. Coding guidelines might not have existed in the early phases of model development or might have changed over time. The model and its variants might grow, increasing entry costs to new coders and model maintenance. Path dependencies might prevent the use of newly emerging data sets, or to integrate methodological advances, or to switch to a better suited software package.

At the same time, the specific grammar of GAMS and GEMPACK largely prevents publicly shared libraries or packages such as those found for Python or R. Instead, communities of economic modellers have gathered around specific implementations of model types, such as in the case of global CGE models around, for instance, GTAP[14], GLOBE[15], or MIRAGE[16]. Each such model has its specific pros and cons to analyse a research question, but picking what is considered best from different models is almost impossible. A key reason is that the building blocks of a model, such as the code for simulation equations, their benchmarking, and reporting of final demand, cannot be easily ported across models as they are not encapsulated. As all symbols have global scope in these AMLs, copy-and-pasting across models leads to name-clashes and the need to harmonise a typically larger number of set, variable and parameter names with the namespace of the receiving model. This has triggered a discussion around more generic modelling platforms which require, for instance, modularity to support different methodological solutions for model parts and/or to add extensions on demand (Britz and Van der Mensbrugghe 2018, Britz et al. 2021b), or the development of modules which can be incorporated in different equilibrium models (Britz et al. 2021a).

4 Summary and recommendations

We argue that a formalised testing strategy is as important in the quality management for an economic model as it is in commercial software. However, differences to industry projects make it difficult to adopt established SQM methods. The typical researcher and their project leader lack the required training on SQM, while their incentives focus on scientific output rather than software stability and maintainability. Moreover, economic models are coded in tailored software packages that lack facilities supporting, for instance, unit testing and automated documentation. Consequently, SQM in economic modelling often turns up late in the life cycle of a model, increasing overall cost for its development and

[14] https://www.gtap.agecon.purdue.edu/

[15] http://www.cgemod.org.uk/globe.html

[16] https://mirage-model.eu/

maintenance. For guidance, we present in annexes two empirical examples of how research teams have implemented SQM for their economic models. We detail the chosen strategy for the bio-economic farm-scale model FarmDyn and the release strategy and related testing of the partial equilibrium model CAPRI. While not all elements of these specific strategies might be relevant or even possible for other economic models, many tools and considerations described should remain useful.

Testing can save considerable costs if it avoids errors which otherwise require that model applications must be repeated or reports partially rewritten. It also increases team members' awareness of quality management more generally. Central testing exposes errors to the whole team. This incentivises each contributor to perform checks during code development, decreasing the number of errors faced by others and detected by central testing. Testing strongly relates to ethics in research by reducing the probability that findings are based on erroneous model outcomes. It supports initiatives to improve the reliability and trust of published research work. For econometric work, some journals require that authors provide reviewers with data and scripts to ensure that reported results can be replicated. The “Journal of Global Economic Analysis”, closely linked to the GTAP community, requires for papers based on model applications, typically of a Computable General Equilibrium model, that one reviewer replicates the runs. Authors are also advised to make their code available as supplementary material. Such initiatives complement testing strategies by modelling teams.

Implementing a testing strategy as described above requires tools such as a software versioning system. Researchers must be trained in these tools, and the project management must ensure their proper usage. The tests itself for the examples described in annex were implemented in GGIG, another tool requiring training; the same will hold for alternatives. Like knowledge of statistical packages or AMLs, training courses on such tools can also be centralised at department level or above. Especially for young researchers, certified participation in SQM related training courses can foster their careers. Given the societal debate about ethics in research, knowledge about and application of SQM related tools that reduce the probability of incorrect research outcomes might soon no longer be a nice-to-have but an essential.

There are increasing demands to document data used in research and to ensure access after the end of projects, as expressed in the ubiquitous requirement to provide data management plans. Similar requirements might arise for software use. Early adopters might benefit from a first-mover advantage by setting standards. As a research community, economic modellers might be well advised to exchange knowledge about the current state-of-the-art to ensure software quality and to jointly develop solutions to be proposed to donors, instead of later facing demands which are ill-fitting to their domain.

A first-time implementation of a testing strategy requires considerable resources, such as licence fees, dedicated hardware to host, e.g., the model repository, and staff training. At least in software development projects, it has been found that these costs pay off, and related savings increase the earlier

errors are detected. The same is likely for economic modelling. Therefore, project managers should encourage regular testing already from the earliest stages of research and development of an economic model.

Funding statement

This study has received funding from the European Union's Horizon Europe research and innovation program as part of the project ACT4CAP27 ('Advancing Capacity and analytical Tools for supporting Common Agricultural Policies post 2027', grant agreement ID 101134874). Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

References

Anzt, H., Bach, F., Druskat, S. et al. (2021). An environment for sustainable research software in Germany and beyond: current state, open challenges, and call for action. *F1000Research* 9:295 https://doi.org/10.12688/f1000research.23224.2

Artaza, H., Hong, N. C., Corpas, M., Corpuz, A., Hooft, R., Jiménez, R. C. et al. (2016). Top 10 metrics for life science software good practices. *F1000Research*, *5*. https://doi.org/10.12688/f1000research.9206.1

Baxter, R., Hong, N. C., Gorissen, D., Hetherington, J., and Todorov, I. (2012). The research software engineer. *Digital Research 2012*. Conference in Oxford, United Kingdom, 10/09/12 - 12/09/12. https://www.research.ed.ac.uk/files/65195747/DR2012_12_1_.pdf (accessed December 17, 2024)

Beck, K. (2002). *Test driven development by example (1st edition)*. Addison Wesley Professional.

Boehm, B. and Basili, V. R. (2007). Software defect reduction top 10 list. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*, 34(1), 75

Britz, W., Kuiper, M. H., Zawalinska, K., and Salvatici, L. (2021a). Increasing model transparency, quality and coherence by deploying tested modules. In *EU Conference on modelling for policy support: Abstracts* (pp. 5-6), JRC127150

Britz, W. and Kallrath, J. (2012). Economic Simulation Models in Agricultural Economics: The Current and Possible Future Role of Algebraic Modeling Languages. In: Kallrath, J. (eds) Algebraic Modeling Systems. Applied Optimization, vol 104. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-23592-4_11

Britz, W., Ciaian, P., Gocht, A., Kanellopoulos, A., Kremmydas, D., Müller, M., Petsakos, A., and Reidsma, P. (2021b). A design for a generic and modular bio-economic farm model, *Agricultural Systems* 191(June 2021): 103133. https://doi.org/10.1016/j.agsy.2021.103133

Britz W. (2010). *The red book on CAPRI coding.* University Bonn, Deliverable D7.1 of the CAPRI-RD project.

Britz W., Lengers, B., Kuhn, T., and Schäfer, D. (2016). *A highly detailed template model for dynamic optimization of farms - FARMDYN*, University of Bonn, Institute for Food and Resource Economics, Version September 2016, 147 pages.

Britz, W. (2014). A New Graphical User Interface Generator for Economic Models and its Comparison to Existing Approaches, *German Journal of Agricultural Economics* 63(4): 271-285. https://doi.org/10.52825/gjae.v63i4.1964

Britz, W., and Witze, P. (2014). *CAPRI model documentation, version 2014*. University Bonn, https://www.capri-model.org/docs/CAPRI_documentation.pdf

Britz, W. and van der Mensbrugghe, D. (2016). Reducing unwanted consequences of aggregation in large-scale economic models - A systematic empirical evaluation with the GTAP model, *Economic Modelling* 59: 462-47. https://doi.org/10.1016/j.econmod.2016.07.021

Britz, W. and van der Mensbrugghe, D. (2018). CGEBox: A Flexible, Modular and Extendable Framework for CGE Analysis in GAMS, *Journal of Global Economic Analysis* 3(2): 106-1763. https://doi.org/10.21642/JGEA.030203AF

Britz, W. and Kallrath, J. (2012). Economic Simulation Models in Agricultural Economics: The Current and Possible Future Role of Algebraic Modeling Languages, in: Kallrath, J. (ed.): *Algebraic Modelling Systems: Modeling and Solving Real World Optimization Problems*, pp 199-212, Springer, Heidelberg, Germany

Christensen, G. and Miguel, E. (2018). Transparency, Reproducibility, and the Credibility of Economics Research. *Journal of Economic Literature*, 56 (3): 920–80. https://doi.org/10.1257/jel.20171350

Exter, M.E. and Ashby, I. (2019). Preparing today's educational software developers: voices from the field. *J Comput High Educ* 31, 472–494. https://doi.org/10.1007/s12528-018-9198-9

Ferraro, P. and Shukla, P. (2020). Is a replicability crisis on the Horizon for Environmental and Resource Economics. *Review of Environmental Economics and Policy* 14(2), pp. 339–351. https://doi.org/10.1093/reep/reaa011

Finger, R., Grebitus, C., and Henningsen, A. (2023). Replications in agricultural economics. *Applied Economic Perspectives and Policy 45(3), pp. 1258-1274.*https://doi.org/10.1002/aepp.13386

Fischetti, M. and Luzzi, I., (2009). Mixed-integer programming models for nesting problems. *Journal of Heuristics*, *15*(3), pp.201-226. https://doi.org/10.1007/s10732-008-9088-9

Green, R. and Ledgard, H. (2011). Coding guidelines: finding the art in the science. *Commun. ACM* 54(12): 57–63. https://doi.org/10.1145/2043174.2043191

Hannay, J. E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., and Wilson, G. (2009) How do scientists develop and use scientific software? *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, Vancouver, BC, Canada, 2009, pp. 1-8, https://doi.org/10.1109/SECSE.2009.5069155

Hermann, S., and Fehr, J. (2022). Documenting research software in engineering science. *Scientific Reports*, *12*(1), 6567. https://doi.org/10.1038/s41598-022-10376-9

Hertel, T.W. and Tsigas, M. (1997). Structure of GTAP. In Hertel, T.W. (ed.) *Global Trade Analysis: Modeling and Applications.* Cambridge University Press. https://www.gtap.agecon.purdue.edu/resources/res_display.asp?RecordID=4840

Hunt, A. and Thomas, D. (2003). *Pragmatic unit testing in Java with JUnit*. The Pragmatic Bookshelf

ISTQB (2024) International Software Testing Qualifications Board. Website, https://www.istqb.org, Accessed on November 24, 2024.

Journal of Global Economic Analysis (2024). Submissions. https://www.jgea.org/ojs/index.php/jgea/about/submissions. Accessed on December 12, 2024.

Kasy, M. (2021). Of Forking Paths and Tied Hands: Selective Publication of Findings, and What Economists Should Do about It. *Journal of Economic Perspectives*, 35 (3): 175–92. https://doi.org/10.1257/jep.35.3.175

Khamis, N., Rilling, J., and Witte, R. (2013). Assessing the quality factors found in in-line documentation written in natural language: The JavadocMiner. *Data & Knowledge Engineering*, *87*, 19-40. https://doi.org/10.1016/j.datak.2013.02.001

Löfgren, H., Lee Harris, R., and Robinson, S. (2002). A Standard Computable General Equilibrium (CGE) Model in GAMS. International Food Policy Research Institute.

Long, F., Mohindra, D., Seacord, R. C., Sutherland, D. F., and Svoboda, D. (2013). *Java coding guidelines: 75 recommendations for reliable and secure programs*. Addison-Wesley

O'Regan, G. (2019). *Concise guide to software testing*. Springer Nature, Cham, Switzerland. https://doi.org/10.1007/978-3-030-28494-7

Pidd, M. (2002). Simulation software and model reuse: A polemic. In *Proceedings of the winter simulation conference*, 1: 772-775. IEEE. https://doi.org/10.1109/WSC.2002.1172959

Podhora, A., Helming, K., Adenäuer, L., Heckelei, T., Kautto, P., Reidsma, P., Rennings, K., Turnpenny, J., and Jansen, J. (2013): The policy-relevancy of impact assessment tools: Evaluating nine years of European research funding, *Environmental Science & Policy* 31: 85-95. https://doi.org/10.1016/j.envsci.2013.03.002

Runeson, P., (2006). A survey of unit testing practices. *IEEE software*, *23*(4): 22-29. https://doi.org/10.1109/MS.2006.91

Searchinger, T., Heimlich, R., Houghton, R.A., Dong, F., Elobeid, A., Fabiosa, J., Tokgoz, S., Hayes, D., and Yu, T.H., (2008). Use of US croplands for biofuels increases greenhouse gases through emissions from land-use change. *Science*, 319(5867): 1238-1240. https://doi.org/10.1126/science.1151861

Soergel, D.A., (2015). Rampant software errors may undermine scientific results. *F1000Research*, *3*. https://doi.org/10.12688/f1000research.5930.2

Storm, H., Heckelei, T., and Baylis, K. (2024). Probabilistic programming for embedding theory and quantifying uncertainty in econometric analysis. *European Review of Agricultural Economics* 51(3), pp. 589–616. https://doi.org/10.1093/erae/jbae016

Vokolos, F. I., and Weyuker, E. J. (1998). Performance testing of software systems. In *Proceedings of the 1st International Workshop on Software and Performance* (pp. 80-87). https://doi.org/10.1145/287318.287337

Westland, J.C., (2002). The cost of errors in software development: evidence from industry. *Journal of Systems and Software*, 62(1): 1-9. https://doi.org/10.1016/S0164-1212(01)00130-3

Zolkifli, N. N., Ngah, A., and Deraman, A. (2018). Version control system: A review. *Procedia Computer Science*, 135: 408-415. https://doi.org/10.1016/j.procs.2018.08.191

Annexes

A1: Strategies for continuous testing in FarmDyn

A1.1 The farm-scale bio-economic model FarmDyn

FarmDyn (Britz et al. 2016) is a farm-scale bio-economic model written in GAMS, depicting farm management in great detail. Typical model configurations in comparative-static deterministic mode comprise 1.000 to 10.000 linear constraints and variables, some of which are integer variables, to yield a Mixed Integer Program for which reliable and fast solvers are available. Model sizes can increase substantially if stochastic, dynamic, or stochastic-dynamic programming is used. Blocks of equations relating to certain farm branches or model extensions, such as for risk or risk behaviour, are integrated on-demand only. This modularity, which breaks the model equations into logically related blocks, keeps the model manageable and speeds up its solution and checking of results. Equally, modules under development can be turned off for production runs. Technically, modularity is mainly realised based on conditional compile-time includes. The code is hosted on a software versioning system and written loosely following coding guidelines developed originally for the partial equilibrium CAPRI model (Britz 2010). FarmDyn was stepwise extended in research projects, following a path discussed in section 2.2. FarmDyn features a Graphical User Interface (GUI) realised in GGIG (Britz 2014); see annex A2.

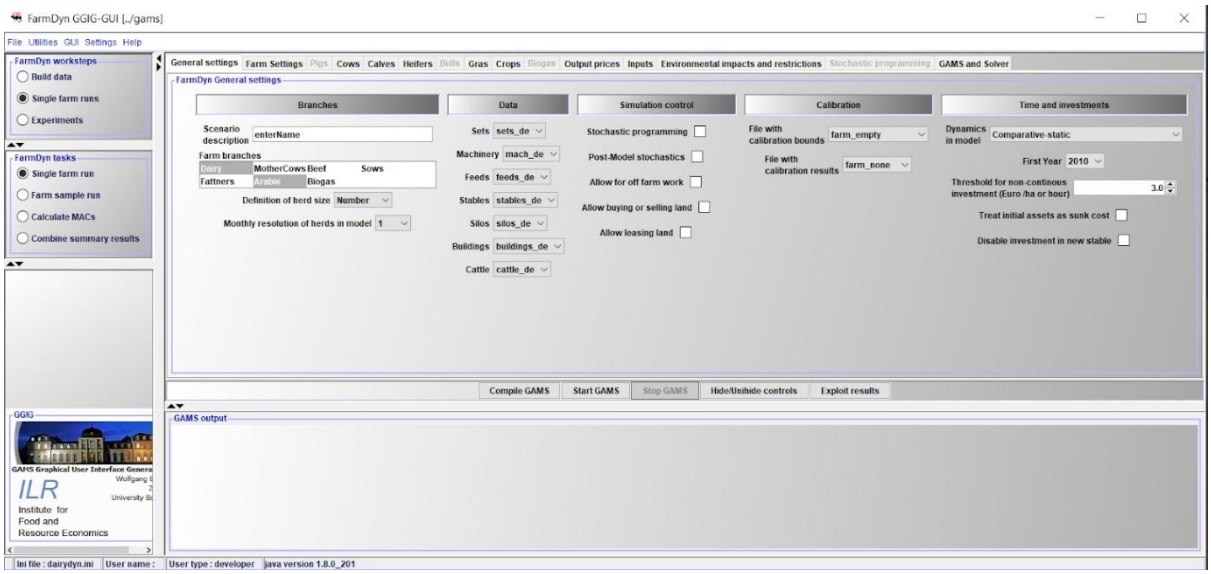
FarmDyn comprises modules for farm branches (arable cropping, dairy, beef cattle, mother cows, fattners, biogas) and depicts in detail related farm management. For instance, labour use is differentiated on a bi-weekly basis and feeding at a monthly one. Investment decisions relate to specific machinery and stables and are based on integer variables to consider return-to-scale.

Past applications encompass of FarmDyn the assessment of GreenHouse Gas abatement options and costs in German and French dairy farm (Lengers et al. 2014, Mosnier et al. 2019), of potential changes in the German biogas program (Schäfer et al. 2017), of the German implementation of the Nitrates and Water Framework Directives on different farm types (Kuhn et al. 2019), of breeding options in German dairy herds (Pahmeyer and Britz 2020), or the interaction of coupled support for legume cropping and the implementation of the nitrate directive (Heinrichs et al. 2021). FarmDyn was also linked to results of crop growth models (Kuhn et al. 2020). More methodological work in the context of FarmDyn related to developing meta-models from detailed farm-scale models, building on large-scale sensitivity analysis (Lengers et al. 2014, Kuhn et al. 2019, Seidel and Britz 2020), and to develop a calibration method based on bi-level programming (Britz 2020).

A1.2 Testing in FarmDyn based on its Graphical Interface Generator

FarmDyn features a Graphical User Interface (GUI) realised in GGIG (Gams Graphical Interface Generator, Britz 2014), a package coded in Java which combines an interface generator for GAMS or R based projects with a report generator (Britz et al. 2015). Besides FarmDyn, GGIG is used for other economic models, such as the partial equilibrium model CAPRI (Britz and Witzke 2014) and the computable general equilibrium model CGEBox (Britz and Van der Mensbrugghe 2018), both mainly developed at the University Bonn, the partial equilibrium model PEM (OECD 2011) and the general equilibrium model METRO (OECD 2015), and the single-farm model IFMCAP at the EU's Joint Research Center (Louichi et al. 2018). The technical solutions detailed next related to testing for FarmDyn draw on features of GGIG and can therefore also be easily applied to these other models. A screenshot of the entry page of the GUI for simulation runs can be seen below in Figure A1.1.

Figure A1.1: FarmDyn GUI



In batch mode, GGIG produces HTML-pages for each started process as shown below in Figure A1.2

Figure A1.2: HTML page with results from test run generator by GGIG

GGIG batch execution report

Runs are generated from	**C:\cgebox\gui\testCompile.txt**
at	**Sa, 16 Jan 21 11:26:56**

Workstep	*Task*	*User*	*Directories: work result dat restart*	*Settings*	*Gams options*	*Started Ended Time used*	*RC*	*Listing GdxDiff*
Simulation	Simulation		C:\cgebox\gams ../results ../data -	**All settings in use**(click to show/hide) **Non default settings in use**(click to show/hide)	compile --scen=com_inc --ggig=on -threads=8 -procDirPath=c:\scrdir optdir=opt	Sa, 16 Jan 21 11:26:57 Sa, 16 Jan 21 11:26:58 0 min and 0 secs	0	c:\scrdir\16.01.2021_11.26.56 \1.lst
Simulation	Simulation		C:\cgebox\gams ../results ../data -	Use pre-defined configurations=true **All settings in use**(click to show/hide) **Non default settings in use**(click to show/hide)	compile --scen=com_inc --ggig=on -threads=8 -procDirPath=c:\scrdir optdir=opt	Sa, 16 Jan 21 11:26:58 Sa, 16 Jan 21 11:26:59 0 min and 0 secs	0	c:\scrdir\16.01.2021_11.26.56 \2.lst
Simulation	Simulation		C:\cgebox\gams ../results ../data -	Dynamics=Recursive dynamic **All settings in use**(click to show/hide) **Non default settings in use**(click to show/hide)	compile --scen=com_inc --ggig=on -threads=8 -procDirPath=c:\scrdir optdir=opt	Sa, 16 Jan 21 11:26:59 Sa, 16 Jan 21 11:26:59 0 min and 0 secs	0	c:\scrdir\16.01.2021_11.26.56 \3.lst

Source: Screenshot from HTML-Page generated by the GUI of FarmDyn in Batch mode

An automated reporting of differences can be added to the HTML based reports, as shown in Figure 3, usually applied to a smaller vector of key model outcomes. If activated, the HTML page reports the number of records where differences larger than a predefined threshold are found, in the (artificial) example shown in the upper panel of Figure 3 below these are 23 cases at the chosen threshold of 1%. These cases can be inspected directly in the HTML page as shown in the lower panel. If changes are considered acceptable, the test is considered successful. Otherwise, the information which indicators changed by how much might already provide useful information to find and correct errors.

Figure A1.3: Automated reporting of differences in GGIG

Single farm runs	Single farm run		C:\dairydyn\gams ../results ../dat -	**Updated settings**(click to show/hide) **All settings in use**(click to show/hide) **Non default settings in use**(click to show/hide)	run --scen=incgen/runInc --ggig=on procDirPath=c:\scrdir -threads=8 optdir=opt	So, 17 Jan 21 10:18:16 So, 17 Jan 21 10:18:23 0 min and 6 secs	0	c:\scrdir\17.01.2021_10.18.06\2.lst c:\scrdir\17.01.2021_10.18.06\diff_2.gdx p_sumRes # of diffs > 1.0% : 23(click to show/hide)
Single farm runs	Single farm run		C:\dairydyn\gams ../results ../dat -	**Updated settings**(click to show/hide) **All settings in use**(click to show/hide) **Non default settings in use**(click to show/hide)	run --scen=incgen/runInc --ggig=on procDirPath=c:\scrdir -threads=8 optdir=opt	So, 17 Jan 21 10:18:16 So, 17 Jan 21 10:18:23 0 min and 6 secs	0	c:\scrdir\17.01.2021_10.18.06\2.lst c:\scrdir\17.01.2021_10.18.06\diff_2.gdx p_sumRes # of diffs > 1.0% : 23(click to show/hide) WinterWheat,dif1,40,60 WinterWheat,dif2,36,46 WinterRape,new,16,40 profit,dif1,29278,44 profit,dif2,208230,15 NSurplus,dif1,-57,59 NSurplus,dif2,10,17 PSurplus,dif1,-13,68 PSurplus,dif2,0,32 totalLabour,dif1,1081,50 totalLabour,dif2,7200,00 labourUse,dif1,0,15 labourUse,dif2,1,00 margArab,dif1,1123,84 margArab,dif2,962,46 MaizSil,old,20,54 gras2_12_graz100,old,4,13 gras4_14_4cuts_sil100,old,35,87 cows,old,138,51 gras,old,40,00 NorgAppl,old,162,49 labourBoundMonthly,old,3,00 margGras,old,740,29

Source: Screenshot from HTML-Page generated by the GUI of FarmDyn in Batch mode

Like other GUI generators, GGIG generates user-operable controls from textual definitions that primarily define admissible input (ranges). In interactive mode, a user configures the model via these controls and starts it. Results are explored via the so-called listing file or reports provided by the GGIG

report generator. This interactive mode is not suited for systematic testing. A tester would need to manually change settings for many controls according to a document defining each test and wait for each run to finish before checking results, an error-prone and tiring process. Therefore, tests use the so-called batch mode, which automatically runs the model based on control settings predefined in a text file. For each run, the settings used and the return code from GAMS are stored on an HTML page (see Figure A1.2 above). The batch mode can either be started via a GUI dialogue or deployed from a command prompt or other software, which enables automated testing. Input files for the batch mode can be created by copy-and-paste from files generated by the GUI in interactive mode and support program flow structures such as loops and if-conditions to render them more compact. The GGIG batch mode can automatically compare results across code versions if these are stored as parameters in so-called GDX containers. GDX is a proprietary, binary format of GAMS for which application programming interfaces in different programming languages are available. The two GDX containers with the old and new results are compared by the GDXDiff utility from GAMS, called from within GGIG. It produces as outcome a parameter in a third GDX container which reports the differences between the two result sets, subject to a user chosen threshold. GGIG then reads this GDX container and formats its content as HTML code added to the HTML report.

The input choices of the controls offered to the user also span the potential test range of inputs for the economic model. This enables automated tests, subject to the curse of dimensionality if many controls, each with multiple settings, are present. The batch mode of GGIG can generate automatically test instances from the available input choice for certain types of controls. These tests build on the default setting for all controls on the interface. In a loop, for each single selection control such as a checkbox, all potential settings are subject to a test, documented on the HTML page. Afterwards, the control is reset to its default and the settings for the next control are tested. Controls which define numerical input (such as sliders or spinners) are normally not subject to such tests, as introducing a different number in the code is unlikely to provoke errors in compile-time tests. It is however possible to define besides the default registered with the control a second numerical value for a test. This is useful if the code treats, for instance, a zero different from a non-zero value. Equally, controls can be excluded from testing, for two reasons. First, the GAMS code itself comprises some tests; they throw an error at compile time for certain combinations of input settings which are considered illegal. Including these cases would show failed tests, which is actually not true as the user cannot execute the model with these settings. Second, the GUI control definitions comprise so-called dependencies. Choosing between risk behavioural models, to give an example, is only possible for the user if the stochastic version of the model is used. With the deterministic version being the default, tests of the risk behavioural models are not possible and therefore excluded from testing. Such cases require defining additional tests manually, as discussed below.

To develop a test strategy, the frequency of code changes must be reflected, after which testing is necessary. A clean checkout of FarmDyn encompasses about 500 files, all added at some point in the past the first time to the repository and probably changed multiple times later. Such changes are called "commits," and experience shows that commit activity is not equally distributed over time but shows peaks. On average, around 170 commits are taking place each year; the maximum number so far encountered was around 400 in 2018. This peak year reflects the generation of a so-called "stable release" where changes from various projects were re-integrated into a common master version.

A1.3 Layered test approach in FarmDyn

Testing in FarmDyn is based on pre-defined tests of the three types discussed: compile-time, run-time, and outcome tests, see also table A1.1 below. These tests require increasing efforts regarding computation time, setting up the tests, and controlling their outcomes, as discussed next.

A1.3.1 Basic layer – compile-time tests

Compile-time tests with FarmDyn typically take less than a second, so many are performed. The HTML pages generated by GGIG indicate failed runs in red to spot failed ones quickly. Compile-time tests are mainly defined in an automated manner from the GUI controls. They automatically reflect changes in GUI definitions, equivalent to changes in the model's potential input data set. For Farmdyn, these fully automated tests currently comprise about 150 different input sets. They refer partly to different model configurations, such as comparative static versus different types of dynamic runs. They also comprise technical options that should not affect outcomes, such as switching listings on and off. A few manually defined test instances have been added to these automated tests. They complement cases not captured by testing each control individually with all others at their default values.

A1.3.2 Intermediate layer – run-time tests

The intermediate layer comprises run-time tests where key farm management choices such as herd sizes and crop shares are fixed to a known optimal solution for the input data generated based on previous projects. These tests aim to exclude that changes in model structure and default parameterisation provoke infeasibilities. Fixing key management choices will also reduce solving time, as only a few integer solutions match given herd sizes and crop acreages. The GAMS code will throw an error if the model is integer or otherwise infeasible, such that these failed test instances can be easily detected. Another advantage of these tests is that key indicators such as profits or Greenhouse Gas emissions are unlikely to change for a fixed core farm program. Larger changes in such indicators provide a rather sure indication of some flaw in recent changes in the code if the former results were deemed correct. Run-time tests are developed jointly by the team.

A1.3.3 Evolved layer – outcome tests to check numeric results

The final tests check the plausibility of the model outcome on a carefully selected number of total optimisation runs without fixed variables. This is expensive as optimising a single test instance can require several minutes due to integer variables. More importantly, deciding if a test has failed, should no run-time error occur, requires a plausibility assessment of simulation results. The tests mainly comprise cases from ongoing projects to exclude code changes in the same or parallel-running projects that lead to unforeseen outcomes in key results. The plausibility assessment focuses on key indicators such as profits, herd sizes, crop acreages, and some selected environmental indicators. They can be directly retrieved from the HTML pages generated by GGIG; see Annex A2. Additionally, the reported key indicators are collected automatically in an EXCEL workbook, with one sheet for each test instance. The different indicators are in the rows, while the columns refer to a revision number tested. This shows how indicator results change along the history of the code with a colouring scheme visualising differences according to relative changes.

Outcome tests implicitly assume determinism, i.e., that the model will produce the same results on a given set of inputs in repeated runs. This is, however, not necessarily the case; see the notes on the stability tests above. Moreover, and most importantly for the case discussed here, MIP problems are usually not solved to full optimality and often run deliberately in non-deterministic mode, such that the solver does not guarantee identical results on the same problem in repeated runs but runs faster. Guaranteed determinism from a solver perspective is defined strictly as technical, not from a conceptual viewpoint. For instance, changing the order in which the equations enter the model might change results even in deterministic mode. The same holds if purely informational equations are added that cannot alter the optimal allocation from a conceptual viewpoint.

What are the consequences of these observations on determinism? First, tests should be run in a defined environment, such as the same software release and hardware, avoiding parallel load. Second, solvers should be used in deterministic mode. Third, if maximal solution times are deemed necessary, exceeding them should trigger a run-time error instead of continuing execution based on intermediate optimal outcomes. These test conditions aim to ensure that differences must stem from code changes (or input data). Testing becomes much more demanding if (almost) identical results should also be guaranteed over a range of software releases and hardware set-ups. This is not discussed here.

A1.3.4 Test frequency and other details

Tests often fail when only parts of locally changed code are committed, or local code is partly not synchronised with the head revision. Sticking to older versions might be necessary, for instance, to avoid the fact that already finalised model runs need to be repeated and re-documented. To avoid inconsistent master versions, coders might not commit files during a project's lifetime. Once a project is ready, all

local files are updated to the newest versions, and files with local modifications are committed in one large block. This tactic reduces problems related to inconsistent, more fine-grained changes. However, it might also lead to very long log entries, which are hard to understand when looking at the history of a single file. Commits changing many files might also mean that all other coders must invest considerable time to deal with merged files or, even worse, conflicts in their local working copies. Moreover, such a strategy makes it hard to relate outcome changes to dedicated code changes. To avoid a model version comprising local modifications being (involuntarily) tested instead of the current master, it is necessary to use a separate so-called clean working copy for tests, kept synchronised with the current head revision. Before tests are run, this local copy must be updated to reflect recent commits.

Table A1.1: Overview of the testing strategy in FarmDyn

	Frequency	# of test instances	Checking efforts
Compile-time tests	Triggered daily if a commit occurred	~150	Very low
Run-time tests	Triggered daily if a commit occurred	~20	Low
Outcome tests	Triggered weekly if a commit occurred	~10	High

Source: The authors

Compile- and run-time tests that do not require manual checks are run frequently; see Table 1. They are automatically triggered[17] each night should a commit have occurred and run on each commit separately. It is the task of the current quality manager (QM) to check each working day if a new HTML page with test results is available. If the page reports failed tests, the QM will check the commit log to determine who performed the commit(s). These team members are informed and asked to correct the errors. They decide then if they debug the problem alone or with the help of colleagues, such as in cases where issues relate to incompatibilities across modules.

Outcome tests that require manual checks are run weekly and should be assessed during the week. A Jenkins process will trigger these tests automatically over the weekend. The current QM will check the resulting HTML page on Monday. The automated GDXDiff output will report changes in key indicators. If this is the case, the QM will consult with the coders who committed during the week to determine if these changes are intended or deemed implausible and, in the latter case, who will deal with the problem.

[17] Using a Jenkins process, see www.jenkins.io

References in A1

Britz, W. (2014): A New Graphical User Interface Generator for Economic Models and its Comparison to Existing Approaches, *German Journal of Agricultural Economics* 63(4): 271-285. https://doi.org/10.52825/gjae.v63i4.1964

Britz, W., & Witze, P. (2014). *CAPRI model documentation, version 2014*. University Bonn, https://www.capri-model.org/docs/CAPRI_documentation.pdf

Britz, W., Pèrez Dominguez, I., Narayanan, G. B. (2015): Analyzing Results from Agricultural Large-scale Economic Simulation Models: Recent Progress and the Way Ahead, *German Journal of Agricultural Economics* 64(2): 107 – 119. https://doi.org/10.52825/gjae.v64i2.1987%0A

Louhichi, K., Ciaian, P., Espinosa, M., Perni, A., & Gomez y Paloma, S. (2018). Economic impacts of CAP greening: application of an EU-wide individual farm model for CAP analysis (IFM-CAP). *European Review of Agricultural Economics*, *45*(2). https://doi.org/10.1093/erae/jbx029

OECD (2011), "Annex D. The OECD Policy Evaluation Model", in *Evaluation of Agricultural Policy Reforms in the United States*, OECD Publishing, Paris https://doi.org/10.1787/9789264096721-17-en

OECD 2015: METRO V1 model documentation, https://one.oecd.org/document/TAD/TC/WP(2014)24/FINAL/en/pdf

Britz, W. (2020): Automated calibration of farm-scale mixed linear programming models using bi-level programming. *Discussion Paper* 2020:4, Download: Discussion Paper.

Heinrichs, J., Jouan, J., Pahmeyer, C., Britz, W. (2021): Integrated assessment of legume production challenged by European policy interaction: A case-study approach from French and German dairy farms, *QOpen* 1(1): 1-19. https://doi.org/10.1093/qopen/qoaa011

Kuhn, T., Enders, A., Gaiser, T., Schäfer, D., Srivastava, A., Britz, W. (2020): Coupling crop and bio-economic farm modelling to evaluate the revised fertilization regulations in Germany, *Agricultural Systems* 127(C). DOI: 10.1016/j.agsy.2019.102687

Kuhn, T., Schäfer, D., Holm-Müller, K., Britz, W. (2019): On-farm compliance costs with the EU-Nitrates Directive: A modelling approach for specialized livestock production in northwest Germany, *Agricultural Systems* 173: 233-243. https://doi.org/10.1016/j.agsy.2019.02.017

Lengers, B., Britz, W., Holm-Müller, K. (2014): What drives marginal abatement costs of greenhouse gases on dairy farms? A meta-modelling approach, *Journal of Agricultural Economics* 65(3): 579–599. https://doi.org/10.1111/1477-9552.12057

Mosnier, C., Britz, W., Julliere, T., De Cara, S., Jayet, P.-A., Havlik, P., Frank, S., Mosnier, A. (2019): Greenhouse gas abatement strategies and costs in French dairy production, *Journal of Cleaner Production* 236: 117589. https://doi.org/10.1016/j.jclepro.2019.07.064

Pahmeyer, C., Britz, W. (2020): Economic opportunities of using crossbreeding and sexing in Holstein dairy herds, *Journal of Dairy Science* 103(9): 8218-8230. https://doi.org/10.3168/jds.2019-17354

Schäfer, D., Britz, W., Kuhn, T. (2017): Flexible Load of Existing Biogas Plants: A Viable Option to Reduce Environmental Externalities and to Provide Demand-driven Electricity?, *German Journal of Agricultural Economics* 66(2): 109-123

Seidel, C., Britz, W. (2020): Estimating a Dual Value Function as a Meta-Model of a Detailed Dynamic Mathematical Programming Model, *Bio-based and Applied Economic* 8(1): 75-99. https://doi.org/10.13128/bae-8147

A2 Testing model releases in CAPRI

CAPRI is a comparative static agri-food partial equilibrium model combining several linked models and comprising an extensive set of reports of economic and environmental indicators (Britz and Witzke, 2014). Model development started in 1994, and since then, the group of developers has changed, data sources were replaced, while software developments in the GAMS language and supporting tools such as Java along with methodological progress implied larger code changes. The model's code base evolved over dozens of research projects, following the processes described in section 2.2. Consequently, the system today contains heterogeneous code in terms of style, documentation and software features. Large, if not most, parts of the code were developed by people no longer involved. The model code is open to all developers, including a few seasoned modellers from the early days, some analysts at governmental agencies or the European Commission, and several PhD students and research assistants. As a network, they share the model based on public releases to remedy the fuzzy versioning problem (see section 3.3 of main paper).

In response to the challenges of dealing with code developed over more than 25 years and continuing parallel developments in projects, the network of CAPRI developers has established since 2016 a *stable release* cycle that revolves around *major releases*, *minor releases*, and *supported features*. Any stable release is a version of the model, including the associated raw data, that (i) does not change and that (ii) has been subjected to an extensive set of tests. A major release is created after significant updates to the raw data or introduction of new key features into the model, provided that the network has the required resources.

Technically, the release cycle is based on tags and branches in the versioning control system, illustrated in Figure 3. The creation of a new major release, such as Stable Release 2 (STAR 2), starts with generating a dedicated branch for this release from the trunk of the code repository (revision 6629 in Figure 3). On this branch, developers prepare code for the major release, for instance, by removing unfinished developments present in the trunk and fixing bugs found during testing and later use. Besides bug fixes, continuing code developments in the trunk and other branches do not find their way into the STAR branches.

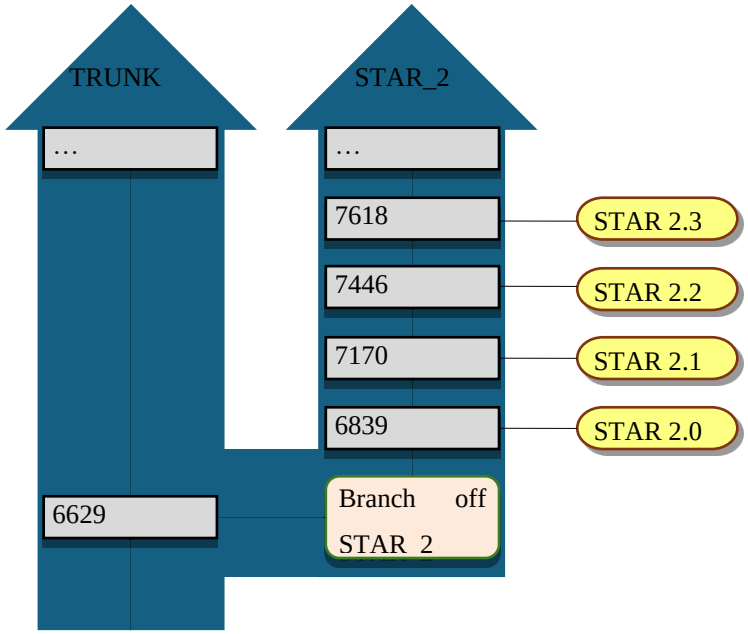
A release and follow-up sub-versions must pass a test suite of *supported features* such as a list of shocks, model configurations, and data constellations agreed upon by the CAPRI network. Technically, a batch execution file for the GUI is set up – using the same techniques as described in the example of FarmDyn above – that carries out run-times tests of database preparation steps, baseline forecast, benchmark calibration, and selected scenario simulations, with features to support in these different steps systematically turned on and off. As a basic stability test, the test suite is run on two different machines with two versions of GAMS. Moreover, the tests are repeated with and without starting values from existing solutions or a pseudo-random number generator and are also repeated without any change.

The results are pairwise compared to find numerical instabilities. Tests of release 2.7 of CAPRI assessed the stability of 5.3 million numbers per simulation with the core model under different hardware and GAMS versions and with and without a previous existing solution, which defines starting values in some sub-processes. The tests revealed that results indeed depend to an astonishing degree on the hardware and software versions: 65% of the results were unstable within the numerical precision limits of GAMS. However, just 0.61% showed deviations larger than 0.1%, while 0.15% of the results were sometimes zero or non-zero. Interestingly, the existence of starting values had no impact at all, but running the process twice with the very same software and hardware changed 35 numbers (0.0007%) by up to 10%. This probably reflects that other processes running on the machine affect the overall computing load and influence the number of repeated solves of certain processes due to time limits.

No systematic outcome tests are performed in the supported features, albeit many such tests are included in the various processing steps. In the terminology of section 3, the integration tests are manually triggered.

The first successfully tested version is “tagged” in the versioning software with a new version number, such as STAR 2.0 in revision 6839 in Figure 3. Despite the tests, subsequent use will typically reveal some bugs. Once a (subjectively) sufficient amount or severity of bugs is fixed, a new minor release is tagged, such as STAR 2.1 in revision 7170 (after 47 commits – the remaining 284 commits between 6839 and 7170 were to other branches or the trunk) in Figure 3. As the major release STAR 2 aged, less errors were found leading to fewer yearly commits, decreasing the frequency of minor releases and of tedious integration tests. Due to the version control system, all tagged releases remain “perpetually” available, facilitating the reproduction of previous studies and, for instance, the external review of the model and its outcomes.

Figure 3: Branching and tagging releases in CAPRI, by the example of STAR 2. Numbers in grey boxes are the incremental revision numbers created by Subversion at each code commit. Rounded yellow terminators represent “tags”, i.e., released versions.



Source: The authors

References in A2

Britz, W., and Witze, P. (2014). *CAPRI model documentation, version 2014*. University Bonn, https://www.capri-model.org/docs/CAPRI_documentation.pdf