Automation Nation: Taming Complex V&V Workflows

Erik Rosenlund[1,2] Robert Hällqvist[1,2] Robert Braun[2] Petter Krus[2]

[1]Saab Aeronautics, Linköping, Sweden
[2]Fluid and Mechatronic Systems, Linköping University, Sweden

Abstract

In the realm of model verification, ensuring traceability and repeatability is paramount for achieving Modeling & Simulation (M&S) credibility and development efficiently. This paper explores challenges and solutions for expressing complex model Verification & Validation (V&V) workflows that ensures readability while still enables automation, parallelization and advanced customization of verification activities. This paper contributes a practical solution that is evaluated through a detailed implementation within an in-house developed multi-purpose automation and Verification & Validation (V&V) framework.

Keywords: Model Verification, Machine-Interoperable Traceability, Workflow Automation, Model-driven Engineering, Simulation Credibility, Verification and Validation

1 Introduction

To quote Balci and Ormsby (2000): “any use of a model in a context where it can not be proved to be credible is by definition not credible”. Credibility is, to say the least, a broad concept involving many aspects of model development and use (*Standard for models and simulations* 2024). As systems and models become larger and more complex (Ahmed, Shah, and Umar 2016), so does the need to establish model credibility (Arthur et al. 1999). Where previously a few verification scenarios were executed and verified manually (Arthur et al. 1999), there is now a move towards extensive model exploration (Bouskela et al. 2023) with increasingly complex, multistep verification activities being implemented (Rosenlund et al. 2024).

In several ongoing industrial research projects, there is a large focus on traceability and frameworks such as System Structure and Parameterization (SSP) Traceability and the Credible Simulation Process (Ahmann et al. 2022). Where traceability of verification activities was previously based on manual documentation, as verification activities increase in size and complexity, this approach is no longer plausible (Arthur et al. 1999). The growing number of experiments increases the need to automate the verification process to maintain development efficiency. And just as in research, one of the core pillars for motivating trust in results is the reproducibility of experiments (Dalle 2012), where traceability of data, both regarding the conducted activity and its results, enables decisions to be taken on "known knowns" and "known unknowns" (Nallaperumal and Krishnan 2013).

To address these needs, the primary research question is formulated as follows:

> **Research question**
>
> How can automated model verification be conducted in a manner that is both traceable, repeatable and comprehensible?

Our proposal couples a lightweight workflow schema based on the Extensible Markup Language (XML) with Git-based change management to satisfy both repeatability and traceability (Figure 1). Because every transformation and activity are explicit in the workflow and intermediate artifacts are version-controlled, stakeholders can audit exactly how a simulation result was obtained. In proposing a schema for representing complex model verification workflows, we build upon insights from existing solutions used in automated data processing while adhering to requirements specific to model verification. Furthermore, evaluation of existing change management methods seek to enhance traceability throughout the verification process; potentially leveraging a mature ecosystem for data management.

To enable close collaboration between industry and academia, the established method denoted as "Industry-as-laboratory" by Potts (1993) was utilized. A number of use cases, established from industrial needs, were used for evaluation of different model verification and exploration

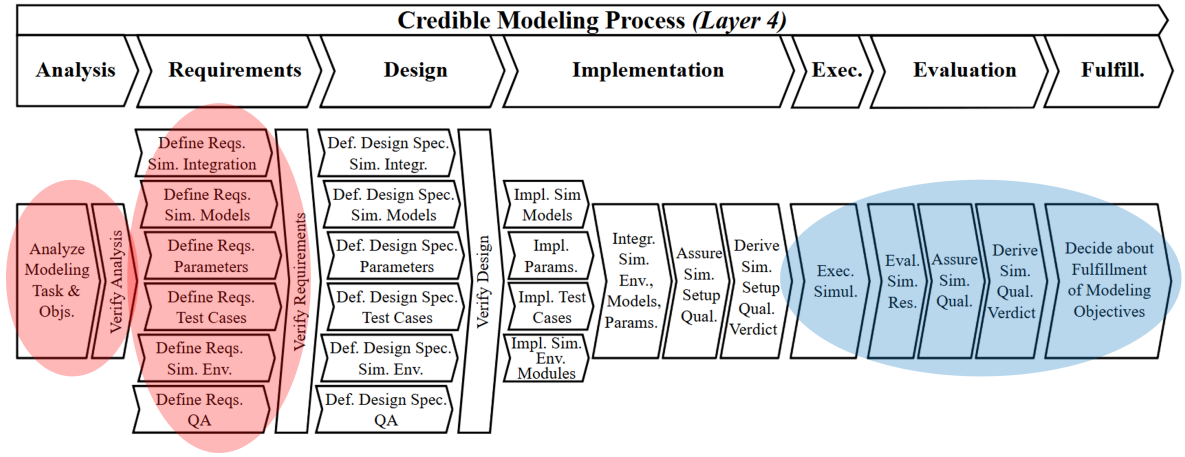

Figure 1. Visualization of the Credible Modeling Process Layer of the Credible Simulation Process Framework (Modelica Association 2022). The red areas showing the verification design phase and the blue area the verification execution phase

scenarios in an in-house automation and testing framework. The focus is here on verification, but the solution can equally well be applied to validation, e.g. as described in (Hällqvist 2023). Looking ahead, our findings offer promising avenues for further refinement and extension in future research, addressing ongoing challenges and exploring new opportunities in model management and V&V.

Contributions This study contributes to the advancement of model V&V through the integration of both novel and established solutions, aiming to promote greater transparency in the process. This by introducing a traceable and repeatable workflow representation—as a simple XML schema—that promotes readability and enabling automation.

The work is demonstrated within an in-house V&V automation framework, integrating established version control methods within model archives. It explores aligning model change management with existing software practices and the SSP Traceability framework, offering a scalable approach to managing changes and enhancing traceability and automation both during and after initial model development.

2 Theoretical Background

This section introduces the key frameworks and concepts that support the development and traceability of simulation models. These include the Credible Simulation Process Framework, version control systems, automated verification and validation mechanisms, and data pipeline architectures. Together, they form the basis for ensuring model credibility, enabling traceable workflows, and supporting scalable, reusable simulation practices.

Credible Simulation Process Framework and SSP Traceability To ensure that design decisions are based on credible data, SSP Traceability (which acts as an enabler of the Credible Simulation Process) was developed. This research focuses on a set of activities that are part of the Credible Modeling Process layer within the Credible Simulation Process. As illustrated in Figure 1, the Credible Modeling Process is largely analogous to the model development and delivery process used at Saab Aeronautics (Andersson and Carlsson 2012). The activities highlighted in blue in Figure 1 are addressed in this work, while those marked in red provide essential artifacts that serve as prerequisites for the activities under focus. A subset of these artifacts is discussed in (Rosenlund et al. 2024).

The Credible Simulation Process Framework allows the tracking of modeling activities, digital artifacts, and decisions to provide a solid foundation for model credibility assessment tasks. It provides the means of documenting data related to simulation tasks in a transparent and comprehensive manner (Modelica Association 2022). The related metadata is stored alongside the model using mechanisms implemented in SSP for coupling metadata related to the model within the same container. It employs various specialized metadata formats, such as Simulation Task Meta Data (STMD) for tracing tasks and Simulation Resource Meta Data (SRMD) for resource data.

DOI 10.3384/ecp218751

Version Control System Extensive usage of Version Control System (VCS) is employed by software companies to maintain collaboration and manage evolving products (Khleel and Nehéz 2020). They are used to storing code, documentation, and related artifacts in a traceable manner, where each change is accompanied by metadata detailing its relationship to other versions, the date, and the author (Khleel and Nehéz 2020). A transition from Centralized Version Control System (CVCS), such as Apache Subversion (SVN), to Decentralized Version Control System (DVCS), e.g. Git, is in progress (Muşlu et al. 2014), largely to enable more fine-grained control during development. The key benefit, in our context, of this transition is that it enables using VCS for tracking model changes even when the model is packaged within archives, as is the case for Functional Mock-up Interface (FMI) and SSP.

Verification and Validation Automation Framework An internal automation framework for model verification and validation, which supports extensive model exploration—the term here referring to the activity of subjecting the model to multiple scenarios while its behavior is checked against requirements or validated against test data (Hällqvist 2023)—serves as an application playground for this research. Model exploration in this context serves to strengthen model credibility thus enabling efficient and large scale model development, model-based decision-making in any life-cycle phase, and improve anomaly detection in model or physical product. The framework provides mechanisms for documenting both model and simulator Operational Domains (ODs) and Domains of Validation (DoVs)[1]. The DoV are all operating conditions where the model has been tested, including outcomes of verification or validation activities, and it serves as a foundation for assessing various aspects of the model's *credibility* (*Standard for models and simulations* 2024). In contrast the OD (or *applicability domain*) defines where the model can be used in regard to a set of requirements, including acceptance criteria, that reflect the model's intended use. A result of this is that the OD will always be a subset of the DoV, since requirement fulfillment outside the DoV is per definition unknown. More detailed descriptions of mode ODs and DoVs can be found in the literature, see for example Oberkampf and Roy (Oberkampf and Roy 2014) and Beisbart et al. (Beisbart and Saam 2019). Establishing both ODs and DoVs is crucial for ensuring the appropriate and credible application of a model, promoting traceability with respect to verification activities and facilitating model reuse by clearly communicating model limitations (Rosenlund et al. 2024).

Data Pipeline / Workflow Frameworks A data pipeline—or workflow—is based upon the architectural pattern **pipes and filters**, in this case more specifically the **tee and join** variant, where filters correspond to activities and pipes to the chaining of these activities (Buschmann et al. 1996). In short, pipes are a set of activities arranged in a specific order, where inputs and outputs are connected between the activities, often creating a Directed acyclic graph (DAG). The pattern's main advantage, when constructing verification workflows, is that "changes should be possible by exchanging processing steps or by recombination of steps, even by users" (Buschmann et al. 1996). The pattern also provides determinism regarding execution order and data flows; from the activity Directed acyclic graph (DAG), parallelization opportunities can be identified. Due to the often computationally expensive operations related to running simulations, this can result in considerable performance gains.

There are a number of existing software solutions that employ this pattern in various forms; two examples of more specialized solutions are **Jenkins** (*Jenkins (software)* 2025), for setting up build flows, and **dask** (*Dask (software)* 2025) for data processing and computation workflows. These solutions employ various methods for expressing workflows, using specific Domain-specific languages (DSLs), general programming languages such as python or existing data storage solutions such as JavaScript Object Notation (JSON) or XML.

3 Methodology

The methodology adopted in this work was driven by the need to define a verification workflow representation that could balance traceability, repeatability, readability, and automation possibilities. The foundation of the implementation lies in a set of verification activities that must be supported, as well as in the coupled requirements governing the verification process. The activities and requirements, collected from verification engineers, are in several cases supported by other sources to strengthen their validity (Cederbladh, Cicchetti, and Suryadevara 2024; Andersson and Carlsson 2012). The resulting methodology is presented in

[1] Or Domain of Verification

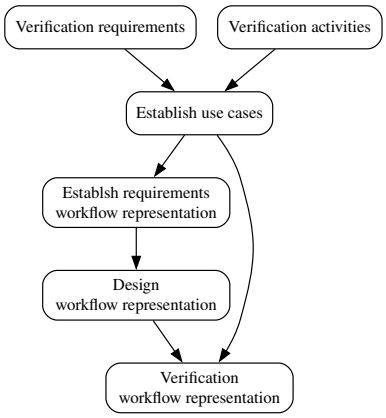

Figure 2. The research process employed

Figure 2, which illustrates the research and development process: starting with the collection of key verification activities and requirements, moving toward well-defined use cases, and culminating in a fully specified, validated workflow.

Requirements

- **Traceability** - Create traceability regarding verification activities and its results, striving for alignment against SSP Traceability
- **Repeatability** - Repeatability of simulation results, this implies that any operation must be deterministic
- **Automation** - The workflow needs to be automated to streamline development and enable effective regression testing
- **Human and machine-readable** - To facilitate both manual design and verification together with a high level of automation, any workflow representation must be both human understandable and machine-readable
- **Customization** - To enable reuse of similar operations between models
- **Resource utilization** - Computationally expensive operations require parallelization to ensure a quick feedback loop during V&V and development

Verification activities

- **Custom activity** - Easy implementation of verification activities
- **Activity comparison** - Compare the outcome of different activities, including simulation results for different scenarios or parameter sets
- **Result aggregation** - Allow aggregation of results

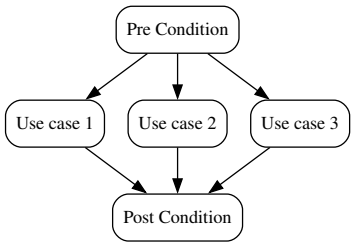

Figure 3. Multiple use cases sharing a common pre and post conditions

Table 1. Mapping between verification activity and use cases

	UC1	UC2	UC3
Activity Comparison	x		
Result aggregation	x	x	
Model Exploration		x	x
Design Optimization			x
Design of Experiments			x

- **Model exploration** - Enable methods of model exploration to evaluate different scenarios or parameter sets
- **Design optimization** - Provide means of using iterative optimization methods such as Bayesian or evolutionary algorithms
- **Design of experiments** - Evaluate what experiments would provide the most value in regard to what we currently know, often implicitly fulfilled by **Design optimization** since this is the first step when using iterative optimization methods

3.1 Use case

Several different use cases were formulated to capture the functionality needed to support the verification activities while fulfilling the defined requirements. Due to the broad nature of the activities, creating multiple smaller use cases enables a better coverage of the activities while also enabling a tighter connection to the final verification, see Figure 3. A mapping between the verification activities and use cases is in Table 1 ensures coverage.

Pre Condition The cases start with a model and a need to verify, validate or optimize certain aspects of the model or system.; they all begin with the establishment of the verification workflow representation itself where all activities are defined and it is decided what additional artifacts are to be saved for traceability. This representation is stored within the model archive for traceability and enabling repeatability, in alignment with SSP Traceability.

DOI 10.3384/ecp218751

Post Condition The tasks all have a final activity where the additional traceable artifacts relating to the verification are packaged within the model (FMI) or system (SSP) container, this to enable a strong traceability connection between the verification activity and data to the model. The post condition states that there should be a containerized model that can be utilized and simulated.

3.1.1 Application Use Case 1

Description The first use case is designed to evaluate multiple product designs over a number of test scenarios. A real world example may be when evaluating different pipe sizes in a system during different load conditions. The process begins by establishing what test scenarios are to be evaluated. For each test scenario, multiple model variants are created and simulated, and the results are subsequently compared. Since all simulation variants are independent of one another, they can be executed in parallel.

Scenario

1. Load existing scenarios from a file, store the scenarios or its location
2. Create the different design variants, store meta data of the design variations
3. Simulate the different variants in parallel
4. Collect and compare the Quantities of Interest (QoIs), store the results associated with the QoIs

3.1.2 Application Use Case 2

Description The second use case is designed to evaluate different behaviors of a model, this by creating multiple combinations of input stimuli and evaluating the results. This is often applicable as verification after model adjustments or prior to deployment to verify that the model is fully functioning. May involve using a number of preset scenarios or creating randomized permutations of scenarios based on methods such as Monte Carlo, then evaluate the resulting behavior in regard to one or multiple requirements and finally generate a OD that corresponds to the fulfillment of that requirement. Just as in the first case, the generated scenarios are not dependent on each other and all can be executed in parallel. To fulfill the requirement of repeatability, any randomization based method must either store the full scenario or metadata enabling recreation of the full scenario. Metadata could for example include reference to the software used for scenario generation, randomization method and/or seed.

Scenario

1. Create the different scenario variants, store meta data regarding the scenarios
2. Simulate the different variants in parallel
3. Collect the QoIs, evaluate and aggregate the fulfillment of the requirement against the DoV as an OD, store the resulting OD using Simulation Task Meta Data (STMD)

3.1.3 Application Use Case 3

Description In the third case, the goal is to find the optimal design of a geometry under some predefined constraints by using Bayesian methods for optimization, where the outputs of previous simulation are used to find better sampling positions. May be applicable in cases were complex interactions between models make optimization challenging, for example in optimizing cooling capabilities in relation to usage scenarios. Using iterative methods creates difficulties when it comes to determinism and repeatability of results. Synchronous batch-sequential methods have previously been employed in other areas using Bayesian search methods (Tran et al. 2022). In these the simulations are grouped in batches and when a batch is completed the next one is allowed to start. This allows each batch to utilize the outcome of previous batches to calculate new sampling positions while ensuring determinism. The main downside of this method is that it is often less effective than using methods that do not take batching into consideration, since the variance of the model simulation time can greatly influence the utilization ratio of the computational resource.

Scenario

1. Store the optimization algorithms and corresponding parameters, variables and, if applicable, randomization seed
2. Create the batch of design variants to evaluate based upon known information, store meta data of the design variations
3. Simulate the batch in parallel
4. Collect the QoIs and evaluate if the design goal is fulfilled, if not return to Item 2
5. Store the parameter set of the final design

3.2 Requirements of Workflow Representation

The workflow representation is to provide clarity as to what, and in which order, any verification

activity is to be executed. It should help maintain a clear trail of artifacts, enabling determinism, traceability and repeatability. To enable complex scenarios, it should support nested workflows, where sub-workflows have a clear connection to parent activities. Moreover, it should enable the execution of independent tasks concurrently, whether locally or in a distributed environment such as High Performance Computing (HPC), to optimize resource usage and efficiency.

The representation should have the ability to encapsulate common verification actions, such as simulation runs or result comparisons, into reusable components, to reduce code duplication, enhance maintainability, and lower the risk of human errors. To further enhance the possibility for re-use of components, it should possess the capacity for easy customization of activities, such as test conditions, parameter sweeps, and simulation scenarios. The workflow should also enable adaptation to intermediate results by using conditional rules that change the next steps based on factors like missing artifacts or whether goals were met. Finally, it should allow for comments, fostering clarity and collaboration by documenting the purpose and logic of each part of the workflow.

3.3 Design of Workflow Representation

We identified existing solutions for representing verification workflows and considered established domain-specific languages and workflow engines, such as Jenkins, Dask, Nextflow, Prefect, Luigi, Airflow, and Snakemake. While these tools offer advanced parallelization and dependency management features, our analysis came to the conclusion that their complexity and overhead often exceeded what was necessary for this scope.

Instead, we landed in a simplified Application Programming Interface (API) to express these tasks and dependencies. This approach maintained the option to layer on more sophisticated workflow management tools in the future, should complexity or scalability demands increase. Once the interface was established, the next decision was how to represent the workflows themselves: using an existing General-purpose language (GPL), such as python or develop a custom Domain-specific language (DSL) in the form of existing markup languages such as xml or json. Using a GPL offers the benefit of leveraging a mature ecosystem with readily available tools. However, it also requires that users are proficient in the chosen language and capable of effectively make use of existing tools. To keep complexity down and simplify access, both to users and potential no code/low code applications, the solution landed on limiting supported features though the utilization of an existing markup language. But to cater the more advanced users, it was also made possible to exploit the API directly as showcased in the Appendix under Listing 10 in cases where access and readability to non developers is deemed less important.

Selection of markup language began by evaluating common existing key-value pair data formats, including YAML Ain't Markup Language (YAML), JSON, Tom's Obvious, Minimal Language (TOML) and XML to determine how they fit the requirements. The final selection of XML was guided by three main considerations. First, the chosen format had to be easily interpretable and editable by engineers, ensuring that the workflow could be understood, maintained, and customized without invoking a specialized programming knowledge. Second, the storage solution needed to supply flexibility and be extensible to support a diverse range of verification tasks, including dependency management, parameterization, and dynamic adaptations. Third, we prioritized easy to use parsing possibilities and schema validation capabilities, all of which contributed to improved maintainability and scalability. Fourth, the selected solution should enable interoperability with traceability frameworks sush as SSP Traceability. To ensure that the verification workflow remains accessible and robust, we imposed guidelines on the structure and complexity of the XML files. Using standardized field naming conventions, encouraging descriptive comments and employing schema validation provides immediate feedback and clarity on correctness.

3.4 Version Control

To enable traceability of the evolving model and not just the final version, VCS—specifically Git—was integrated into existing model packaging formats such as FMI/SSP archives. In this use case, version control and packaging standards serve **orthogonal purposes**. The solution does not change the internal structure of the archive, it merely adds data for tracking changes. The combination a symbiosis; embedding a repository inside an archive enables tracking changes, ensures reproducibility, and allows auditing of the lineage of verification tasks. This approach used familiar version control mechanics, enabling seamless collaboration, comparison of different versions, and

DOI
10.3384/ecp218751

storage of historical results for review.

3.5 Verification of Workflow Representation

Finally, the methodology was verified through tests that confirmed its ability to meet the stated requirements. The approach was stress-tested by scaling the number and complexity of verification scenarios according to the use cases, with observations showing stable performance, clear error-handling mechanisms, and effective parallelization strategies. These tests also examined the impact of version control integration on model archive size and complexity.

4 Results

The solution consists of two distinct parts: first, the markup schema for defining the verification workflow, allowing for easy customization and repeatability of verification activities, its inherent portability making it optimal for providing traceability by integration into FMI or SSP archives; and second, a python function library—contained within the in-house V&V automation framework—defining the verification activities, providing a static point that multiple workflows can operate against.

4.1 Resulting Workflow Representation

The structure and attributes of the resulting XML schema are described in this section along with the connection to the initial requirements.

Action Starting with the smallest artifact of the schema, the link to the automation framework library of predefined functions. It specifies a named reference inside the library, any additional fields within the scope are custom parameters that will be passed to the library function as seen in Listing 1. A function together with its parameters effectively creates templates for simplified re-use to reduce code duplication and maintenance overhead. Instead of rewriting actions for similar tasks, engineers can adapt templates by altering a few parameter values, making the verification process more flexible, modular, and cost-effective.

Listing 1. Activity example

```
<action function="simulation">
    <parameter_1 value="f"/>
</action>
```

Grouping mechanisms Due to the often computationally expensive operations involved in executing simulations, the speed-up factor of parallelizing very slow activities—such as simulations—can often be close to optimal since any overhead will almost always be much smaller than the actual simulation. Therefore, enabling parallelization is essential for scalability, where multiple simulations should be processed simultaneously. To facilitate this, two different grouping types were introduced: **sequential** (Listing 2) and **parallel** (Listing 3). These grouping types enable clustering of activities, the former defining a dependency chain of execution in sequential order and the latter that all activities within the group are independent and could be executed individually.

Listing 2. Sequential example, creating a sequential dependency chain between the actions, forcing execution in a specific order

```
<sequential>
    <action function="action_1"/>
    <! -- potential additional actions -->
</sequential>
```

Listing 3. Parallel example, all actions can be invoked independently of each other

```
<parallel>
    <action function="action_1"/>
    <! -- potential additional actions -->
</parallel>
```

Structure The overall structure is based on two main principles. First, a hierarchical structure that enables nested grouping operations, allowing for complex scenarios while maintaining coherence and clarity (see Listing 5). The second principle is dynamic adaptation, which allows any action to change or add actions downstream.

Dynamic adaptation has two main objectives, the first being to facilitate adaptation to unknown factors. For example, an action that searches for scenarios in a folder has no way to know how many, or what kind of, scenarios it will find before execution. To enable this, the action, after finding the scenarios, will make adjustments to downstream actions to reflect the new information, Listing 5 showcases a workflow that corresponds to this scenario, simulating and verifying results for each input data file it finds in a directory at runtime, the definition of the function called shown in Listing 4. The second objective of dynamic adaptation is to enable the workflow to take into account intermediate results such as failures or fulfillment of conditions and automatically branch to alternative paths. This adaptability is often connected to handling iterative tasks where the number of iterations may not be known beforehand.

Listing 4. Python signature of `find_files`

```
def find_files(path: str,
               pattern: str = "*.csv") ->
                   list[Node]:
    """Creates corresponding nodes to
        availible files that match *pattern
        * under *path*."""
```

Listing 5. Dynamic adaptation, adapting to the number of files found

```
<sequential>
  <action function="find_files">
    <path value="file_storage/"/>
  </action>
  <sequential>
    <action function="simulate"/>
    <action function="evaluate_results"/>
  </sequential>
</sequential>
```

Combining these two principles enable the building of almost any workflow, including generating scenarios and design variations, comparing results, and executing different actions depending on the outcome, all while maintaining the readability of both simple and complex workflows. In the end, the basis for enabling these requirements, while keeping the representation compact and understandable, was to outsource much of the complexity to the automation framework's function library; this by creating helper functions for advanced use cases. The ability to construct templates of common verification scenarios also creates reusability opportunities and limits code duplication between workflows.

Global parameters Outside the scope of the workflow, it is possible to place **global parameters**, as seen in Listing 6. Any number of global parameters can be created, and they will be propagated to all functions, this to enable some common understanding of context without explicitly feeding the parameters, for example model location, to each action. As seen in Listing 6, the top workflow definition consist of a single sequential or parallel grouping.

Listing 6. Workflow example

```
<VerificationWorkflow>
    <model value="./model_1"/>
    <sequential>...</sequential>
</VerificationWorkflow>
```

Limitations The order of execution is strictly downstream, meaning that each action can only be executed once in a specified sequence. This results in that an action should not create a downstream dependency to any action that is upstream. Data, on the other hand, can only be accessed upstream where a dependency exists to ensure a deterministic result. Figure 4 showcases how requirements regarding execution order, parallelization opportunities and data access relate.

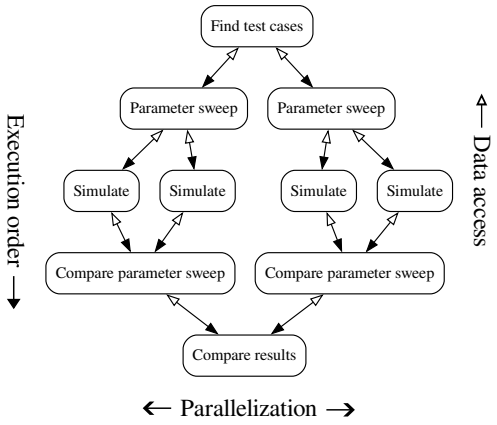

Figure 4. Showcasing execution order, parallelization possibilities and data flow

To mitigate some complexities related to merging branches executed in parallel, there is a defined limit to the change scope for dynamic adaptations. An action may only adjust actions or groups that are within its own grouping scope. For example, in Listing 7, **outer_action_1** can alter everything downstream, including the inner actions. However, when moving into the inner scope, the **inner_action_1** can only alter actions downstream within its grouping scope, more specifically **inner_action_2**.

Listing 7. Dynamic adaptation scope

```
<sequential>
    <action function="outer_action_1"/>
    <sequential>
        <action function="inner_action_1"/>
        <action function="inner_action_2"/>
    </sequential>
    <action function="outer_action_2"/>
</sequential>
```

4.2 Implementation

The V&V framework parses the workflow representation and constructs a corresponding Directed acyclic graph (DAG). Any actions grouped by a clustering mechanism are enclosed by start and end nodes to facilitate simpler graph manipulations. This design is chosen for the same reason that most programming languages use notations such as {} or tabs to clarify the beginning and end

DOI 10.3384/ecp218751

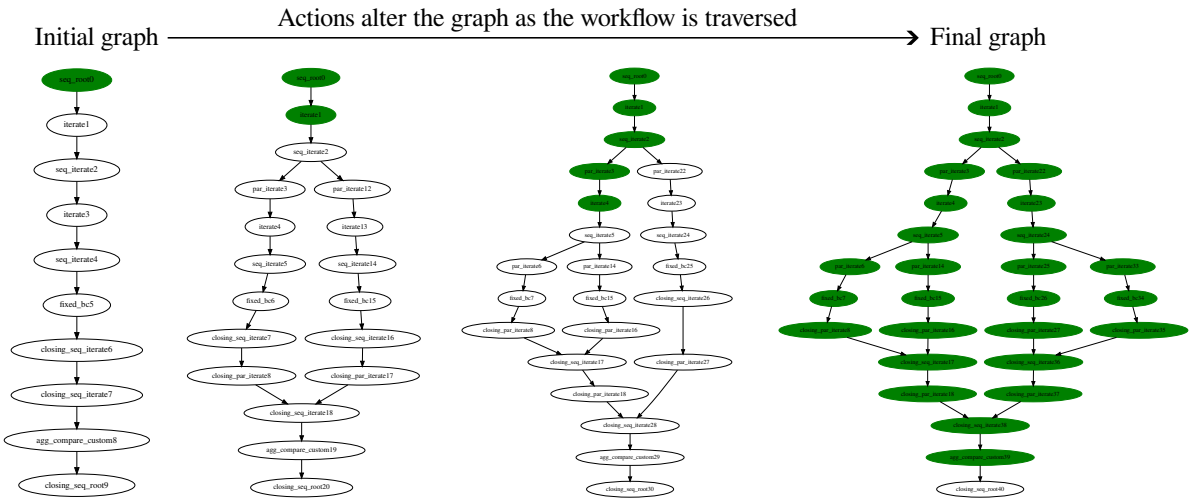

Figure 5. Graph adaptation during execution for use case 1. Filled node marks completed actions

of constructs. Although the start of a construct can often be inferred from context, it is often challenging to distinguish where the construct ends without a closing notation.

After finding the starting node, the application begins traversing the DAG downstream, morphing the graph as the actions execute, see Figure 5 for a description of how the graph alters after that actions are processed. To enable this, the full DAG is passed to each action for modification and when returned, it is merged back into the original DAG before continuing execution. The graph corresponding to use case 1 is showcased in Figure 5; the traceable artifacts of the activity and workflow representation are described as SSP Traceability artifacts in the Appendix, see Listing 8 and Listing 9.

4.3 Version Control

By integrating version control systems within model archives, each verification activity and its results could be linked to specific model versions, parameters, and configurations, thereby creating a comprehensive audit trail, see Figure 6 for a visualization of the concept. This enhanced traceability facilitated root-cause analysis in cases where discrepancies arose and provided a robust foundation for conducting credible and reproducible verification processes. But despite these successes, some limitations surfaced. Handling large binary artifacts within the version control system was challenging, inflating repository sizes and complicating storage. Future improvements might involve adopting specialized large file storage solutions or more effective strategies.

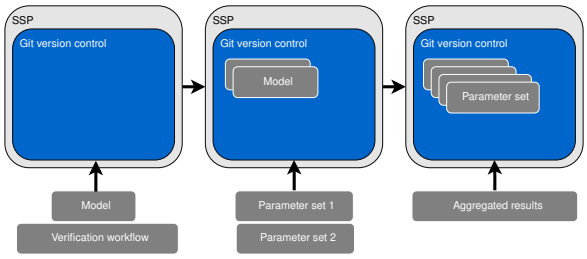

Figure 6. Showcasing the version control methodology, each step a snapshot in time enabling traceability of model creation and alterations

4.4 Verification

When applied to the use cases, the workflow consistently produced traceable and reproducible verification results. Stakeholders perceived the enhanced traceability as increasing the trustworthiness of verification outcomes. Preliminary evaluations suggested that the new workflow format offered clearer logic navigation and easier interpretation than previous solutions. The workflow representation was perceived to minimize the need for code duplication between projects. Additionally, the options for using templates for common functions to further simplify the workflow design are promising and warrant further evaluation.

5 Discussion

Workflow representation By comparing the verification method to established practices highlights some improvements. Traditional methods, such as the *Saab Aeronautics Handbook for Development of Simulation Models* (Andersson and Carlsson 2012), often rely on non machine readable, document-centric tracking, which may be error-prone and sometimes difficult to maintain. By consolidating workflow definitions and

model data into a consistent, machine-readable form, the proposed approach reduces human error, enhances discoverability, and aligns well with growing industry trends toward continuous integration and continuous delivery.

The function library contained within the automation framework offloads much of the complexity from the workflow representation and direct manipulation of the DAG, enabling a simpler representation of the workflow. Offering fine-grained control via arguments or flags enabled teams to adapt the workflow to various conditions without creating entirely new test definitions. However, this raises questions whether it might sometimes be too simplistic. For example, in Listing 5, it is not possible to determine how the function **find_files** will alter the downstream actions by only looking at the representation. This places requirements on the function library to provide clarity regarding what functions do and how they affect the workflow; Underscoring the need for intuitive defaults, comprehensive documentation, and clear guidance to ensure broad and consistent user adoption. It is identified that a possible mitigative action is to foster collaboration best practices regarding workflow adaptations among users.

During the design, the discussion rose early to include the possibility to let workflow designers work directly in python, skipping the markup schema. In some cases one is preferred, in others not. It almost always comes back to how to enable easy access for all stakeholders, not only developers, to the verification workflows. We identified that XML would align more closely with standards such as SSP and FMI but may in some cases be quite explicit provide less readability for use cases such as this. If a port to some other format would be deemed necessary in the future, XML is highly convertible into almost any other key-value pair format.

Implementation By breaking large verification tasks into parallel segments, total execution time was reduced; however, coordinating these tasks introduced new challenges in synchronizing data states and results. Many of these challenges relate to how branches should be merged; for example, how to handle situations where two branches have made different changes to future common activities. To cope with this, it was necessary to place restrictions on how dynamic adaptation functions in order to mitigate some complex cases. Prohibiting feedback loops in the graph is one such restriction that minimizes the chance of overwriting artifacts at consecutive invocations. Disallowing loops does lead to some complexity when taking iterative workflows into account, and creative workarounds are sometimes needed to achieve certain complex cases. For example, in iterative workflows to achieve conditional effects the solution could be to copy the desired actions to the end when a condition is triggered, thus creating the next iteration.

The chosen implementation solution for parallelization has created a high level of complexity when coupled to the ability for dynamic adaptation of the graph. The design decision not to use a central graph handler to orchestrate the parallelization, but to let parallel threads traverse the graph independently of each other, enables a very high degree of parallelization. In hindsight, most likely unnecessarily high since any simulation activities will be magnitudes slower, a clear case of premature optimization. The implication of the design is that each parallel branch has its own notion of the DAG. Without a central authority, the branches must negotiate the new notion of truth before continuing when merging branches.

Traceability The intention originally was to utilize Simulation Resource Meta Data (SRMD) for storing and the resulting workflow definition. After delving deeper into the definition of SSP Traceability, it was later adjusted to utilizing an external resource together with STMD. Both SRMD and STMD are connected to the **Credible simulation process**, but more fitting would be to have a stronger connection to the **credible modeling process** by utilizing Modeling Task Meta Data (MTMD). But at the moment this is not included into the released version of the standard and the content between the metadata formats are in large interchangeable, thus the choice went to using the released STMD

Embedding version control systems directly within the model archives offers a transparent, maintainable history that aligned well with established practices such as FMI/SSP. The resulting benefits, including easier reproduction of results, better understanding of model evolution, and stronger stakeholder confidence, generally outweighed the overhead and complexity of managing multiple branches, commits, and binary data within a repos-

DOI
10.3384/ecp218751

itory. Data management and storage considerations emerged as another critical aspect of the solution. Storing binary artifacts within version control systems allowed for a single source of truth and integrated traceability, but also increased repository sizes, potentially affecting performance and long-term maintainability. Strategies like introducing binary files early, employing artifact repositories or Git LFS, and limiting unnecessary duplication can mitigate these issues. Nonetheless, as verification activities scale up, organizations will need to carefully balance repository complexity against the benefits of having all artifacts co-located and tracked within the same environment.

6 Conclusion

This work has presented a method for managing model verification activities that meet the stated research objective of achieving traceability and repeatability in verification workflows, while ensuring readability and enabling automation. By preserving the complete history of verification activities, providing a transparent and evidence-based record that in the end supports more informed decision-making during and after development. Binding verification logic closely to model versions, harnessing parallel execution, and embracing dynamic adaptation, the approach supports more rigorous and scalable verification scenarios. By integrating traceable verification workflows with version control embedded into model archives, the approach provides a platform for managing complex and evolving verification processes.

The Modelica association standards FMI, SSP, and SSP Traceability serve as key enablers to the presented work. Relying on such standards allow the presented functionality to interplay, through portable and stand-alone simulation packages, with similar functionality provided in both open source and commercial tools, such as Dymola (Bruck 2023). Work to initiate this described interplay between testing frameworks and development tools has been initiated in the OpenSCALING project and at Saab through, for example, the work of Matstoms (Matstoms 2025). Matstoms realizes SSP Traceability, and automation, in an existing regression testing framework including model development tools in the loop, particularly Dymola, OpenModelica (Mengist et al. 2015), and easySSP (exxellent Solutions 2025).

The positive reception, strong performance gains, and improved user experience provide a solid foundation for ongoing refinements and broader integration into current software development practices, continuous integration processes, and simulation ecosystems that use standards such as FMI or SSP. As best practices and tools continue to evolve, integrated, traceable, and machine-readable verification workflows are likely to continue evolving and becoming more mature, in the end benefiting a wide range of simulation and modeling applications.

Acknowledgments

This research was conducted within the scope of the Advanced digitization ERATO projects as well as the ITEA4 OpenSCALING project, funded by Vinnova and Saab AB. The authors would also like to thank Dan Louthander for his work with the in-house verification framework.

References

Ahmann, Maurizio et al. (2022-11-22). “Towards Continuous Simulation Credibility Assessment”. In: *Proceedings of Asian Modelica Conference 2022, Tokyo, Japan, November 24-25, 2022*. Asian Modelica Conference 2022, Tokyo, Japan, November 24-25, 2022, pp. 171–182. DOI: 10.3384/ecp193171. URL: https://ecp.ep.liu.se/index.php/modelica/article/view/572 (visited on 2025-03-28).

Ahmed, R., M. Shah, and M. Umar (2016-06-15). “Concepts of Simulation Model Size and Complexity”. In: *International Journal of Simulation Modelling* 15.2, pp. 213–222. ISSN: 17264529. DOI: 10.2507/IJSIMM15(2)2.317. URL: http://www.ijsimm.com/Full_Papers/Fulltext2016/text15-2_213-222.pdf (visited on 2025-04-02).

Andersson, Henric and Magnus Carlsson (2012-12-12). *Saab Aeronautics Handbook for Development of Simulation Models : Public Variant*. 12/00159. Linköping University Electronic Press. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-86281 (visited on 2025-03-26).

Arthur, J.D. et al. (1999-12). “Verification and validation: what impact should project size and complexity have on attendant V&V activities and supporting infrastructure?” In: *WSC’99. 1999 Winter Simulation Conference Proceedings. ’Simulation - A Bridge to the Future’ (Cat. No.99CH37038)*. WSC’99. 1999 Winter Simulation Conference Proceedings. ’Simulation - A Bridge to the Future’ (Cat. No.99CH37038). Vol. 1, 148–155 vol.1. DOI: 10.1109/WSC.1999.823064. URL: https://ieeexplore.ieee.org/document/823064/?arnumber=823064 (visited on 2025-04-02).

Balci, O. and W.F. Ormsby (2000-12). “Well-defined intended uses: an explicit requirement for accreditation of modeling and simulation applications”.

In: *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*. 2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165). Vol. 1, 849–854 vol.1. DOI: 10.1109/WSC.2000.899883. URL: https://ieeexplore.ieee.org/document/899883 (visited on 2024-06-05).

Beisbart, Claus and Nicole J. Saam (2019-04-24). *Computer Simulation Validation: Fundamental Concepts, Methodological Frameworks, and Philosophical Perspectives*. Springer. 1088 pp. ISBN: 978-3-319-70765-5. URL: https://www.ebook.de/de/product/30290750/computer_simulation_validation.html.

Bouskela, Daniel et al. (2023-12-22). "The Common Requirement Modeling Language". In: *Modelica Conferences*, pp. 497–510. ISSN: 1650-3740. DOI: 10.3384/ecp204497. URL: https://ecp.ep.liu.se/index.php/modelica/article/view/960 (visited on 2025-01-08).

Bruck, D. (2023-09). "SSP in a Modelica Environment". In: *Proceedings of the Modelica Conference*. Linköping University Electronic Press. DOI: 10.3384/ecp204711.

Buschmann, Frank et al. (1996-08-16). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Volume 1 edition. Chichester: Wiley. 476 pp. ISBN: 978-0-471-95869-7.

Cederbladh, Johan, Antonio Cicchetti, and Jagadish Suryadevara (2024-03-31). "Early Validation and Verification of System Behaviour in Model-based Systems Engineering: A Systematic Literature Review". In: *ACM Transactions on Software Engineering and Methodology* 33.3, pp. 1–67. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3631976. URL: https://dl.acm.org/doi/10.1145/3631976 (visited on 2025-01-10).

Dalle, Olivier (2012-12). "On reproducibility and traceability of simulations". In: *Proceedings Title: Proceedings of the 2012 Winter Simulation Conference (WSC)*. 2012 Winter Simulation Conference - (WSC 2012). Berlin, Germany: IEEE, pp. 1–12. ISBN: 978-1-4673-4782-2. DOI: 10.1109/WSC.2012.6465284. URL: http://ieeexplore.ieee.org/document/6465284/ (visited on 2025-04-04).

Dask (software) (2025-01-12). In: *Wikipedia*. Page Version ID: 1268901352. URL: https://en.wikipedia.org/w/index.php?title=Dask_(software)&oldid=1268901352 (visited on 2025-04-07).

exxellent Solutions (2025). *Orchideo| easySSP, developing and simulating complex systems*. https://www.easy-ssp.com/. (Visited on 2025-07-01).

Hällqvist, Robert (2023). "On The Realization of Credible Simulations: Efficient and Independent Validation Enabled by Automation". PhD thesis. Linköping University, Division of Fluid and Mechatronic Systems. ISBN: 978-91-7929-597-4.

Jenkins (software) (2025-03-11). In: *Wikipedia*. Page Version ID: 1279860802. URL: https://en.wikipedia.org/w/index.php?title=Jenkins_(software)&oldid=1279860802 (visited on 2025-04-07).

Khleel, Nasraldeen and Károly Nehéz (2020-01-01). "Comparison of version control system tools". In: *Multidiszciplináris Tudományok* 10, pp. 61–69. DOI: 10.35925/j.multi.2020.3.7.

Matstoms, Axel (2025). "Automating Modeling and Simulation Verification Testing in a Credible Simulation Process: Leveraging SSP and SSP Traceability". MA thesis. Department of Computer and Information Science, Linköping University.

Mengist, A. et al. (2015-09). "An Open-Source Composite Modeling Editor and Simulation Tool Based on FMI and TLM Co-Simulation". In: *Proceedings of the 11th International Modelica Conference*. Linköping University Electronic Press. DOI: 10.3384/ecp15118181.

Modelica Association (2022-10-27). *SSP Traceability Specification. Version 1.0.0-Beta2*. Standard, unreleased.

Muşlu, Kıvanç et al. (2014-05-31). "Transition from centralized to decentralized version control systems: a case study on reasons, barriers, and outcomes". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: Association for Computing Machinery, pp. 334–344. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568284. URL: https://doi.org/10.1145/2568225.2568284 (visited on 2025-04-03).

Nallaperumal, Krishnan and Annam Krishnan (2013-12-07). *Engineering Research Methodology A Computer Science and Engineering and Information and Communication Technologies Perspective*.

Oberkampf, William L. and Christopher J. Roy (2014-01-13). *Verification and Validation in Scientific Computing*. Cambridge University Press, pp. 378–381. 790 pp. ISBN: 978-0-521-11360-1. URL: https://www.ebook.de/de/product/11824763/william_l_oberkampf_christopher_j_roy_verification_and_validation_in_scientific_computing.html.

Potts, C. (1993-09). "Software-engineering research revisited". In: *IEEE Software* 10.5, pp. 19–28. DOI: doi: 10.1109/52.232392.

Rosenlund, Erik et al. (2024). "Objectively Defined Intended Uses, a Prerequisite to Efficient MBSE". In: *Modelica Conferences*, pp. 29–42. ISSN: 1650-3740. DOI: 10.3384/ecp20729. URL: https://www.ecp.ep.liu.se/index.php/modelica/article/view/1128 (visited on 2025-04-02).

Standard for models and simulations (2024). NASA STD-7009B.

Tran, Anh et al. (2022-04-01). "aphBO-2GP-3B: a budgeted asynchronous parallel multi-acquisition functions for constrained Bayesian optimization on high-performing computing architecture". In: *Structural and Multidisciplinary Optimization* 65. DOI: 10.1007/s00158-021-03102-y.

Appendix

Listing 8. Simulation task metadata, verification task for use case 1

```
<stmd:SimulationTaskMetaData>
  <stmd:EvaluationPhase>
    <stmd:AssureSimulationQuality id="eval_step_01" description="Evaluate simulation results
        against requirements">
      <stc:Input>
        <stc:Resource kind="system" source="../../SystemStructure.ssd" type="application/x-ssp
            -definition"/>
        <stc:Resource kind="testcase" type="application/xml" source="eval_task_01.xml"
            description="Evaluation task definition"/>
      </stc:Input>
      <stc:Procedure>
        <stc:Resource kind="method" type="application/xml" source="procedure.xml" description=
            "Evaluation procedure document"/>
      </stc:Procedure>
      <stc:Output>
        <stc:Resource kind="result" type="application/xml" source="results/evaluation_report.
            xml" description="Evaluation report"/>
      </stc:Output>
      <stc:LifeCycleInformation>
        <stc:Validated date="2025-04-11T09:30:00+01:00">
          <stc:Responsible organization="Team A" name="John Doe" role="Evaluation Engineer"/>
        </stc:Validated>
      </stc:LifeCycleInformation>
    </stmd:AssureSimulationQuality>
  </stmd:EvaluationPhase>
</stmd:SimulationTaskMetaData>
```

Listing 9. Verification workflow use case 1. Referenced above as eval_task_01.xml

```
<VerificationWorkflow>
  <model value="./model_1"/>
  <sequential>
    <action function="find_test_cases">
      <path value="./test"/>
    </action>
    <sequential>
      <action function="parameter_sweep">
        <parameter_name value="parameter_1"/>
        <values value="5, 10, 15"/>
      </action>
      <sequential>
        <action function="simulate"/>
      </sequential>
      <action function="compare_parameter_sweep"/>
    </sequential>
  </sequential>
  <action function="compare_results"/>
</VerificationWorkflow>
```

Listing 10. Verification workflow as Python for use case 1

```
common_parameters = {"model": "./model_1",}

simulate_seq = Sequential(
    Action("simulate")
    )
sweep_seq = Sequential(
    Action("parameter_sweep", parameter="parameter_1", values=[5, 10, 15]),
    simulate_seq,
    Action("compare_parameter_sweep")
    )
top_seq = Sequential(
    Action("find_test_cases", path="./test"),
    sweep_seq,
    Action("compare_results")
    )

result = Test_Framework(parameters = common_parameters, seq=top_seq)
```


DOI
10.3384/ecp218751