To build, or not to build

ModelFlow, a build solution for MDE projects

Beatriz Sanchez
Department of Computer Science
University of York
York, UK
basp500@york.ac.uk

Dimitris Kolovos
Department of Computer Science
University of York
York, UK
dimitris.kolovos@york.ac.uk

Richard Paige
McMaster University, Canada
paigeri@mcmaster.ca
University of York, UK
richard.paige@york.ac.uk
paigeri@mcmaster.ca

ABSTRACT

Conservative execution, end-to-end traceability, and context-aware resource handling are desirable features in model management build processes. Yet, none of the existing MDE-dedicated build tools (e.g. MTC-Flow, MWE2) support such features. An initial investigation of general-purpose build tools (e.g. ANT, Gradle) to assess whether we could build a workflow engine with support for these desirable features on top of it revealed limitations that could act as roadblocks for our work. As such, we decided to design and implement a new MDE-focused build tool (ModelFlow) from scratch to avoid being constrained by assumptions and technical constraints of these tools. We evaluated whether this decision was sensible by attempting to replicate its behaviour with Gradle in a typical model-driven engineering scenario. The evaluation highlighted scenarios where Gradle could not be extended to achieve the desirable behaviour which validates the decision to not base ModelFlow on top of it.

CCS CONCEPTS

• **Software and its engineering** → *Domain specific languages*; *Software maintenance tools.*

KEYWORDS

Model Driven Engineering, Model Management, Workflow, Build tool, End-to-End Traceability

ACM Reference Format:
Beatriz Sanchez, Dimitris Kolovos, and Richard Paige. 2020. To build, or not to build: ModelFlow, a build solution for MDE projects. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410942

1 INTRODUCTION

Conservative execution, end-to-end traceability, and context-aware resource handling are desirable features in model management build processes. Conservative executions ensure that the process only executes required tasks based on the up-to-date state of its resources. Context-aware resource handling minimizes the invocation of potentially expensive load operations. And end-to-end traceability can be used for impact analysis, debugging and identification of refactoring opportunities [6].

To support the build process of MDE projects, some approaches have opted for extending general purpose build tools with support for model management tasks. This is the case of EMF, Epsilon and ATL which have extended ANT to support model management tasks, including model loading, disposal, validation and transformation. While several build tools provide out-of-the-box support for incremental[1] task executions, other features can be more challenging to adapt e.g. using models to influence the task execution order and using them to provide end-to-end traceability. Other approaches such as MTC-Flow [1], MMINT [3] and ChainTracker [5] have opted for dedicated solutions, some placing task-model interdependencies at their core, others managing their traces. However, none of these tools offer the incremental task execution, which is important for efficiency, nor do they offer a context-aware approach for model handling.

We performed an initial investigation of general-purpose build tools such as Gradle [11] and Pluto [4], to assess whether we could build a workflow engine with support for these desirable features on top of it. This investigation (discussed in Sec. 2.2) revealed limitations that could act as roadblocks for our work. As such, we decided to design and implement a prototype from scratch to avoid being constrained by these limitations. Our prototype, ModelFlow, consists of a textual language for specifying model management tasks and their dependencies, along with an interpreter that provides the desirable MDE build tool features.

The semantics and syntax of the ModelFlow language were outlined in a previous paper [13] – without a supporting implementation at that stage. Compared to [13], in this paper we: (a) provide an in-depth analysis of background and related work, (b) present the concrete architecture and implementation of ModelFlow, (c) demonstrate its use in a more comprehensive scenario and at a finer level of detail, and (d) evaluate its capabilities and performance against Gradle.

Having implemented ModelFlow we evaluated whether this decision was sensible by attempting to replicate its behaviour with Gradle (best-of-breed general-purpose build tool) in a typical model-driven engineering scenario. The evaluation highlighted scenarios where Gradle could not be extended to achieve the desirable behaviour; this validates the decision to not base ModelFlow directly on Gradle. Nevertheless, it is recognised that MDE tasks are only a

MODELS '20, October 18–23, 2020, Virtual Event, Canada

[1] In the sense that tasks define and check their inputs to determine if a re-execution is needed

subset of all tasks in a build, and that attempting to develop a complete replacement for e.g. Gradle is a very ambitious task. Thus, in future work we should explore how to 1) either propose extensions to Gradle to accommodate the scenarios we have identified or 2) to provide integration mechanisms between Gradle and ModelFlow.

Roadmap. The rest of the paper is structured as follows. Sec. 2 discusses existing build systems and presents a motivating example that highlights their limitations in the context of MDE processes. Sec. 3 describes ModelFlow and the facilities it provides to address these limitations. Sec. 4 evaluates our solution with an experiment and a qualitative analysis that compares ModelFlow against Gradle. Sec. 5 discusses related work. Finally, Sec. 6 concludes the paper and outlines future work.

2 BACKGROUND AND MOTIVATION

This section starts with a motivating example of a code generation process that illustrates desirable features in a build tool that supports model management tasks. We then discuss and summarize how widely-used and state-of-the-art build tools can support the development process of the example.

2.1 Motivating example

Our motivating example describes a simplified process for generating a Java implementation of a component-based system. The process consists of three model management tasks: validation, model-to-model transformation, and model-to-text transformation. The dependencies between tasks, models, metamodels and file resources are illustrated in Figure 1. For simplicity, all models and metamodels in this example are built with EMF. The *component* model ⓐ represents a set of interconnected component blocks such as the one in Figure 3a . The *configuration* model ⓑ defines Tolerance elements that will be used to filter incoming signals of ports in the *component* model using a configurable tolerance value. The metamodels of ⓐ and ⓑ are presented in Figure 2.

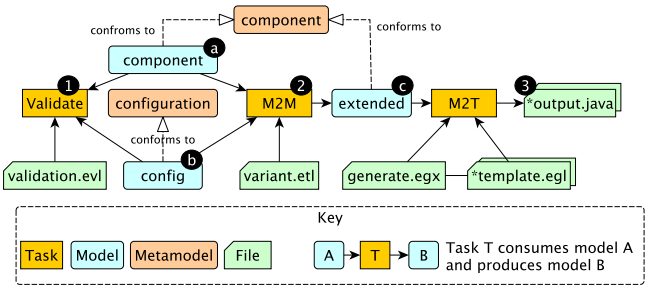


Figure 1: Dependency Graph

The wellformedness of models ⓐ and ⓑ is validated with task ① using EVL [9] invariants[2] such as the ones in Listing 1. The constraint `HasSource` in line 2 checks that all Connector elements in the *component* model have a source, by checking that their `from` property is defined. Similarly, the constraint `PositiveValue` in line 8 checks

[2]The type of element that the constraints act upon and the model they belong to is indicated in the `context` environment e.g. `component!Connector` acts on Connector elements from the *component* model. The `message` is displayed when the check fails.

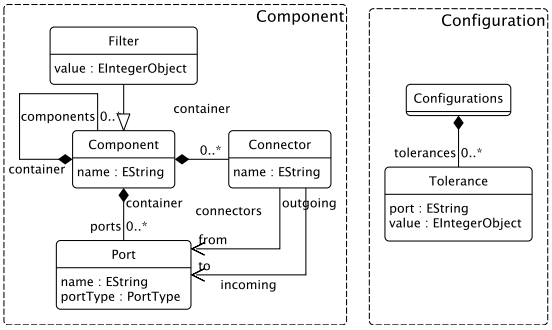


Figure 2: *configuration* and *component* metamodels

that the value of elements of type Tolerance in the *configuration* model are greater than zero.

```
1  context component!Connector {
2    constraint HasSource {
3      check : self.from.isDefined()
4      message : self.name + " has no source" }}
5  context config!Tolerance {
6    constraint PositiveValue {
7      check : self.value > 0
8      message : "Tolerance with no positive value" }}
```

Listing 1: Sample EVL invariants

After the validation step, models ⓐ and ⓑ are consumed by an ETL [8] model-to-model transformation ② to produce the *extended* model ⓒ. The output model is an extended version of the *component* model containing additional Filter components for each Tolerance element in the *configuration* model. Each Filter is populated with the value from the corresponding Tolerance element, and the incoming Connector of the port targeted in the Tolerance is split into two connectors: one that goes into the input port of the Filter and one that comes out from its output port. The Filter is created in the same container as the container of the port targeted by the Tolerance element. Figure 3b represents the corresponding *extended* model that resulted from Figure 3a which contains an additional Filter component represented by the `TemperatureFilter` block.

```
1  public class TemperatureComparator {
2    private Double temperature, targetTemperature,
         difference;
3    private void compute() {
4      /* protected region compute on begin */
5      this.difference = this.targetTemperature-this.
           temperature;
6      /* protected region compute end */
7  }}
```

Listing 2: Generated code of the TemperatureComparator component

The remaining operation is an EGX [12] model-to-text transformation ③ which uses the *extended* model as input to generate Java code. The resulting code establishes the connections between components, but the developer is expected to handwrite their internal logic inside protected regions[3]. These regions are illustrated

[3]A section which should not be overwritten if the model-to-text transformation is re-executed.

in Listing 2 where lines 6 and 8 indicate the start and end of a protected region that can be hand-written.

Provided the dependencies between the model and task artefacts can be specified, a supporting MDE build tool should be able to offer:

- *Conservative task execution.* The build execution must be consistent with the impact that resource changes have on the different tasks. A *conservative* task execution is able to identify and execute only the tasks affected by a set of changes. For example, if we change the `generate.egx` M2T transformation program it would be desirable that the workflow only executes the `M2T` task as everything else is not affected.

- *Context-aware model loading and disposal.* Model management tasks are typically preceded and succeeded by loading and saving/disposing of the models they operate on. Large models can be slow to load and memory-intensive [7, 14]. Therefore they should be loaded only if they are required by tasks of the worklflow. Likewise, to free up memory, it is also important to dispose of them as soon as they are no longer useful. For example, if only tasks `M2M` and `M2T` are to be executed, the *component* and *configuration* models can be disposed from memory immediately after *M2M*'s execution, and the *extended* model does not need reloading to be used by `M2T`. Following the execution of a task, a context-aware model loading and disposal strategy knows if a loaded model needs to be retained in memory to be reused by another task or if it can be safely disposed of.

- *End-to-end traceability.* Capturing traceability links between consumed/produced model elements and/or lines of code can be useful for debugging and analysis purposes. While traceability information is often a by-product of individual model management operations it is rarely offered as a combination of traces from tasks in a workflow. To our knowledge, only ChainTracker [5] offers end-to-end traceability from a workflow. Consider a scenario where a developer noticed an incorrect tolerance value in of the generated code for the `TemperatureFilter` component from Figure 3b. Having access to the traces of the workflow execution could allow the navigation from code, to the *extended* model and then to the *configuration* model where the tolerance value could be fixed.

- *Protection of output resources.* Depending on the scope of a developer's activity, different behaviours could be provided by an MDE build tool. For instance, a developer could be evaluating whether the *extended* model was producing the desired outcome with the generated code. This process could involve manually modifying the model and then executing the code generator. Having a mechanism that prevents the execution of the M2M task (or that at least asks if it should go ahead) can be useful to protect the manual changes in the *extended* model from being overwritten. However, having a mechanism that discards changes made to this intermediate model to restore consistency can be useful in production mode.

2.2 Available solutions

We now discuss the level of support for the desirable MDE build tool features presented above, within widely-used and state-of-the-art build tools: Ant, Gradle, Pluto and Maven. Additional tools such as MMINT, MTC-Flow, MWE2 and ChainTracker will be discussed in the related work section.

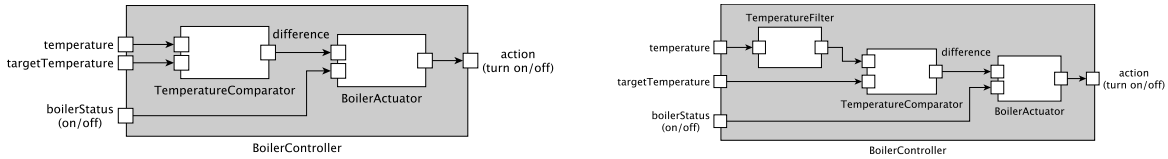
Apache Ant is a widely used build tool written in Java. A build definition in Ant is captured in an XML file and it starts with a root project which contains one or more targets. Each target defines one or more tasks which are sequentially executed. Ant uses target inter-dependencies to compute its execution plan. Ant supports incremental executions through the use of the `uptodate` tag which checks whether a set of target resources are more up-to-date than their source. If this is the case, this element updates the value of a boolean property which can be used by targets to condition their execution.

To illustrate how Ant would support conservative task executions, we use the build script in Listing 3 which triggers a code generation task. The script contains two targets. The first target performs an `uptodate` check (line 3) which updates the boolean value of the `template.updated` property when the `output.java` has a more recent timestamp than the `template.egl` file. In turn, this property is used by the `M2T` target in line 5 as a condition for its execution. This approach enables the triggering of the M2T target only when the template file is *more up-to-date* than the generated file. This strategy has two major drawbacks. One is that up-to-date checks must be explicitly specified for each relevant combination of input and outputs (e.g. transformation script, imported libraries, generated files) that should trigger a given task execution. Specifying this information in both the task specification and the up-to-date checks can be an error prone tasks and it involves duplicating information. Another is that timestamps may not be the right property to determine if files are up-to-date. For example, generated files with protected regions may be modified inside these regions and still be considered up-to-date. A similar reasoning applies if we wanted to protect generated files from being overwritten, in that they would need explicit conditional checks to be setup.

```
1  <project name="Workflow" default="M2T">
2    <target name="checkFiles">
3      <uptodate targetfile="template.egl"
           srcfile="output.java"
           property="template.updated" />
4    </target>
5    <target name="M2T" depends="checkFiles"
         if="template.updated">
6      <epsilon.emf.register file="component.ecore"/>
7      <epsilon.emf.loadModel name="M"
           modelfile="extended.model"
           metamodeluri="Component" read="true"
           store="false"/>
8      <epsilon.egl src="template.egl"
           target=''output.java''>
9        <model ref="M"/>
10     </epsilon.egl>
11   </target>
12 </project>
```

Listing 3: Ant incremental workflow

While Ant does not provide built-in support for model management tasks, relevant extensions have been contributed from tools such as ATL and Epsilon. Examples of Epsilon tasks in Ant are sown in lines 6-8 of Listing 3. In particular, lines 6 and 7 of Listing 3 show how model loading and metamodel registering are two separate tasks that must happen before the model management task of line 8 can use them. Once loaded, these models are kept in



(a) A boiler component **(b) An extended boiler component**

Figure 3: Boiler components

an in-memory model repository which is accessible by all model management tasks. To support context-aware model loading and disposal in Ant, a developer would have to define each task in a separate target and load all the required modes before the execution of a task. Even then, models reused in different Ant targets would have to be disposed and reloaded in each.

Gradle is also a task-based build tool, language and dependency manager. In contrast to Ant, tasks in Gradle do not have to be contained in targets and they can directly depend on other tasks. Its build life cycle consists of three phases: `initialization`, `configuration`, and `execution`. After resolving project dependencies in the `initialization` phase, the `configuration` phase builds a graph of the tasks that are part of the build and computes which of them are required to be executed in the *execution* phase [11].

Gradle was designed to support incremental execution of build scripts. The task execution graph is not only influenced by task interdependencies but also by their inputs and outputs [11]. These values are typically evaluated at the `configuration` phase but some inputs may be `evaluated` at the execution phase [11]. If the inputs of a task have not changed, it is considered up-to-date and skipped, otherwise it is executed [11]. In Gradle, properties of type file, directory or file collections can be declared as inputs or outputs, but properties of arbitrary nature such as strings can only be used as inputs.

As demonstrated later in the paper, despite the ability to declare dynamic task outputs, these are not used to mark tasks as out-of-date. The lack of support for dynamically discovered dependencies [4] makes output protection difficult and can make the computation of the up-to-date status of a task inconsistent with what its input and output resources suggest.

Thanks to Gradle's language extension facilities, custom data structures can be used to declare models which can be accessed by the tasks in the build. An example data structure used to declare the configuration model is shown in the listing below. We discuss this in more detail and with reference to a concrete example in the evaluation (Sec. 4.1).

```
1  models {
2    config(EMF){
3      modelFile = file('resources/m/config.model')
4      metamodelFile =
           file('resources/mm/configuration.ecore') }}
```

Hybrid/pluto is an incremental build tool that performs dynamic analysis to enforce invariants on its dependency graph [4, 10]. This graph connects file nodes with built units (i.e. tasks) through edges that indicate whether the build unit produces or requires the file [4]. The initial version of the algorithm *pluto* [4] interleaved dependency analysis with task execution. The *hybrid* version of the algorithm considered the full dependency graph traversal unnecessary in subsequent executions and proposed the use of file changes to only select potentially impacted tasks and check their consistency to decide whether to re-execute them [10].

To check whether a file is up-to-date, *pluto* uses the notion of *stampers* which are functions which take a file and produce a value or *stamp* based on some criteria such as its last modification date, contents' hash, or existence [4]. These *stamps* are saved in the edges between a file and a task in the dependency graph. Because of the stampers, Pluto is able to offer conservative task execution. Nevertheless, Pluto does not use outputs to determine if a task execution is appropriate.

Apache Maven is a build tool and dependency manager that favours convention over configuration. Maven configures the build process using one or more xml files called POMs.

In contrast to ANT, Maven has three predefined life cycles[4] which go through specific phases in a predefined order. For example its default life cycle includes the phases `validate`, `initialize`, `compile`, and `test` in that order. Invoking any of those phases will implicitly call those that precede it. Custom tasks can be defined but they must be attached to a specific phase of a life cycle. Similarly, the *archetype* of a maven project defines different tasks which are executed by default at different phases of the life cycle. If more than one task is attached to the same phase, they are executed in the order in which they are declared.

While some Maven tasks can execute themselves incrementality, it is not a feature of the build tool.

While some of these tools have support for up-to-date checks to input resources, some are based on timestamps (e.g. Ant) or single mechanisms (e.g. Gradle) and others offer a broad range of possibilities (e.g. Pluto). None of these tools have a task execution mechanism designed to support elements such as models to influence computed plans but there are cases where extensions can be provided so that models influence the execution order (e.g. Ant, Pluto, Gradle). In particular, Ant would requires extensive conditional statements, Pluto would only be able to handle input models and Gradle would need models to be resolved as sets of files. Only Gradle and Pluto use outputs to influence task execution decisions but Gradle can only handle output files and directories when they are known before the task execution and Pluto does not know the outputs until the end of the task's execution. Evidently, none of these tools has support for end-to-end traceability out-of-the-box, but the data structure extension mechanisms of Gradle can be used to capture models declarations which could be used to support end-to-end traceability.

[4] http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

3 MODELFLOW

ModelFlow is a prototype for specifying and executing multi-step workflows involving model management tasks. ModelFlow consists of a textual language for specifying model management tasks and their dependencies, and an interpreter that can conservatively execute such workflows based on changes made to relevant artefacts (e.g. models, model management programs, generated files). ModelFlow also supports context-aware model loading and disposal and offers end-to-end traceability. ModelFlow is in active development and can be found in the EpsilonLabs projects[5].

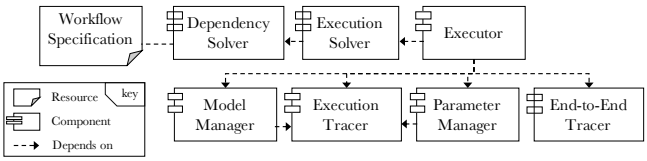


Figure 4: Component diagram of ModelFlow's architecture

The architectural components of ModelFlow are shown in Figure 4. To execute a workflow, ModelFlow receives a workflow specification which contains all the tasks to be executed along with the model resources that they will consume or produce. The workflow specification language is described in Sec. 3.1 using the example from Sec. 2.1. Using the same example, we then describe the build execution process focusing on how it contributes to the conservative task execution (Sec. 3.2), output protection (Sec. 3.3), context-aware resource handling (Sec. 3.4), and end-to-end traceability (Sec. 3.5). Then, we outline some implementation details in Sec. 3.6.

3.1 Workflow specification

The specification of the motivating example workflow is illustrated in Listing 4. A model resource specification requires a name, a model type (e.g. EMF, Simulink) and configuration parameters. For example, lines 2-13 of Listing 4 declare three model resources: *config*, *component* and *extended*. These models are of type `epsilon:emf` and populate the `src` and `metamodelFile` parameters.

A task specification requires a the name of the task, its type (e.g. EOL, ATL, etc.), configuration parameters and the names of any models that are to be consumed, modified or produced. Optionally, a task specification may contain a *guard*, i.e. a boolean condition that should be met for it to be executable; and declare explicit dependencies to other tasks by name. For example, lines 14-28 of Listing 4 declare three tasks: *validate* of type `epsilon:evl`, *m2m* of type `epsilon:etl` and *m2t* of type `epsilon:egx`. These tasks specify their input and output models after the `in` and `out` keywords, respectively, and declare task interdependencies after `dependsOn`. The tasks populate any configuration parameters inside the curly braces environment such as the `src` and `outputRoot` parameters in lines 26-27 of task *m2t*.

The task and model resource *types* will process the parameters in the specification. A task type can declare some of its parameters as inputs or outputs so that tasks can determine if a re-execution is needed in subsequent invocations. For example, the `src` parameter of the three tasks in Listing 4 is declared as an input by the tasks.

[5]https://github.com/epsilonlabs/modelflow

Other task parameters that are not required in the configuration can be implicitly declared as inputs or outputs by the task type. For example the generated files of task *m2t* are declared as outputs by the task type `epsilon:egx`.

```
1  param basedir;
2  model config is epsilon:emf {
3    src : basedir + "config.model"
4    metamodelFile : basedir + "configuration.ecore"
5  }
6  model component is epsilon:emf {
7    src : basedir + "component.model"
8    metamodelFile : basedir + "component.ecore"
9  }
10 model extended is epsilon:emf {
11   src : basedir + "extended.model"
12   metamodelFile : basedir + "component.ecore"
13 }
14 task validate is epsilon:evl
15   in config and component {
16     src : basedir + "validation.evl"
17 }
18 task m2m is epsilon:etl
19   in config and component
20   out extended
21   dependsOn validate {
22     src : basedir + "extended.etl"
23 }
24 task m2t is epsilon:egx
25   in extended {
26     src : basedir + "generate.egx"
27     outputRoot : "src-gen"
28 }
```

Listing 4: ModelFlow workflow

Although this example uses Epsilon tasks and model resources only, our implementation is not bound to this framework neither for tasks nor for resources. ModelFlow's implementation already supports tasks from the GMF and EMF frameworks and it can be non-invasively extended to support additional modeling frameworks and model management tasks.

3.2 Conservative task execution

The first step in the build execution involves deriving a dependency graph from the workflow specification, i.e. a directed graph built from explicit and implicit dependencies between tasks and resources. This is done by the *Dependency Solver* component. Explicit dependencies are created when tasks declare a dependency to another task and when tasks declare which resources are consumed, produced or modified. In the motivating example, the `M2M` task explicitly depends on the `validation` task and M2M explicitly consumes models `component` and `config`. Implicit dependencies are created when task parameters reference another task's outputs e.g. if a task used the generated files form `M2T`.

The next step in the build execution involves transforming the dependency graph into an execution graph which is a directed acyclic graph with task nodes only. This is done by the *Execution Solver* component. The first step of this transformation consists of adding all task nodes from the dependency graph in the new execution graph. The second step consists of adding all task-to-task edges found in the dependency graph. Finally, the algorithm iterates over pairs of tasks creating edges in such a way that they

satisfy the *model resource usage* constraint defined below. In practice this means that task-to-task dependencies take precedence over resource-to-task ones. Attempting to create an edge that introduces a cycle would result in an invalid execution graph and the execution will be aborted.

Definition 1. The *model resource usage* [13] constraint specifies that in an execution, a model resource: (a) can only be produced by one task, (b) can be consumed and modified by any number of tasks, and (c) must be produced before it can be modified.

This constraint ensures that the task graph is built in such a way so that tasks that modify model resources are invoked before tasks that read them, and that tasks that produce models are invoked before those that modify them. In subsequent executions, the interpreter re-uses the same dependency and execution graphs[6].

The next step in the build is to execute the plan. This is orchestrated by the *Executor* component by iterating over the graph in *topographical* order. This ensures all required tasks are executed before the task at hand. In the present implementation of ModelFlow, tasks are executed sequentially. In future work we will add support for concurrent executions.

There are several checks that each task in the iteration needs to perform to decide whether it must run before actually preparing for the execution. These checks are based on the up-to-date state of its input and output parameters and resources. The first check consists of verifying if the task is enabled and if its *guard* is satisfied. If both conditions are met, the task goes on to evaluate its inputs and outputs. If it is its first-time invocation of the task, no further checks are required and the task proceeds to prepare itself for execution. If there has been a previous execution, the *Execution Tracer* component computes whether its output and input models and parameters have changed. A more detailed explanation of how the *Execution Tracer* performs this computation is given in Sec. 3.4. If outputs have changed, the task (or the user) may decide to invoke the output protection logic which prevents the task's execution, or to continue with the checks. If outputs have not changed (or were not relevant to prevent the execution) but inputs have, then the task now prepares for execution.

By the time a task must get ready for execution, its parameters have already been configured but models may still need to be loaded. After models have been loaded, the current input models and parameters are recorded by the *Execution Tracer*. The task is then executed and afterwards their output parameters and models are recorded by the *Execution Tracer*. If available, model management traces can be passed to the *End-to-End tracer* at this stage.

3.3 Protection of Outputs

Consider the *extended* model that is produced from the model-to-model transformation in the motivating example. After executing the workflow, a developer might manually modify this model with the intention of understanding how the changes are propagated to the code using the model-to-model transformation. When a task is being considered for execution, one of the first checks made is whether its output properties and resources have externally changed. In case the outputs have externally changed, if the build execution is in *interactive* mode then it will pause to ask the user whether to execute the task, as this would discard the external changes. If the build execution is in *non-interactive* mode, then the user can specify the desired behaviour in advance by specifying whether to protect or discard external changes.

In the example where the *extended* model is externally modified, triggering a build with the *output protective* behaviour enabled would result in skipping the model-to-model transformation to avoid the *extended* model from being overwritten, and on the execution of the code generation task as its input model would have externally changed. In contrast, triggering an execution with this protective behaviour disabled, would result in the execution of the model-to-model transformation to restore the consistency of this model.

3.4 Context-aware resource handling

In ModelFlow, a *model* is a resource that needs to be loaded into an in-memory representation[7] and be disposed of when no longer in use. In addition, models are resources that influence the task execution order when used across multiple tasks. Both models and task parameters can be used as inputs or outputs of a task. In practice, this means that based on changes to their values from previous executions, a task can determine if its execution is required.

The *Parameter Manager* and *Model Manager* components are both in charge of determining if parameters or models have changed from previous executions. They do so by computing *stamps* for these elements which are then recorded by the *Execution Tracer* component. Stamps were first introduced by Erdweg et al. [4] as values that could precisely indicate whether the file was up-to-date e.g. a timestamp, a hash, etc. Depending on whether the parameter is an input or an output its stamp is computed differently in ModelFlow. For models, the stamp also depends on whether it has already been loaded or not.

Parameter Manager. When the *Executor* starts to process a task and there is a trace from a previous execution available, the *Parameter Manager* is requested to compute new stamps for the input and output parameters. Consider the model-to-text transformation from the motivating example. The *epsilon:egx* M2T task type declares the transformation script (`src`) as an input parameter and the generated files as implicit output parameters. The *Parameter Manager* starts by evaluating the task's input parameters in order to compare their hashes to the ones in the trace. In the case of the model-to-text transformation the input stamp of the *src* parameter is computed as the hash of the file and its value is compared to the one from the execution trace.

The *Parameter Manager* then moves on to assess whether the output parameters have changed from the previous execution. In contrast to inputs, outputs cannot be fully evaluated at this point, unless the task is executed. As such, the trace of output parameters must have sufficient information so that the stamp can be recomputed without having to re-execute the task. For instance,

[6] There is currently no mechanism to invalidate the graphs if a workflow specification has changed

[7] The data that needs to be loaded in memory depends on the underlying modelling framework/tool. For example, an XMI-based EMF model needs to be fully loaded into memory, while a database-backed model persisted in NeoEMF [2] or CDO [15] can be loaded partially and on demand.

the stamp value for the generated files is a map that uses the path of generated files as keys and their stamp as value.

The implementations for the different types of tasks can contribute their own parameter stampers. For example, the *epsilon:egx* task type contributes one that computes the stamp from the contents of the file, ignoring any text within protected regions.

If the task is to go ahead with the execution, the already computed stamps for the input parameters are registered by the *Execution Tracer*. After the task is executed, the parameter manager computes a new stamp for the output parameters and the *Execution Tracer* records the new values.

Model Manager. After processing the input and output parameters, the *Model Manager* goes on to evaluate the stamps for input and output models if a previous execution trace is available. As for task parameters, the *Model Manager* uses different stampers depending on whether models are inputs or outputs. In addition, the stamp of input models is computed differently if the model has been loaded or not. More precisely, the *unloaded stamp* is computed for output models and for input models that have not yet been loaded e.g. for being used by another task. If the input models are available as loaded models, then the *loaded stamp* is computed instead. Consider the *configuration* and *extended* models used by the model-to-model transformation from the motivating example. In this case, *configuration* is an input that was used in the previous validation task and *extended* is an output. As such the stamp for the *extended* model is calculated using an unloaded stamper, while that for the *configuration* model is computed using a loaded one. Both of these models are of *epsilon:emf* model type which contributes both a loaded and an unloaded stamper. For these model types, the loaded stamp is computed by serializing (not persisting) the in-memory EMF resource while the unloaded stamp is computed from the file. A similar approach can be adopted for models split across many files, computing the stamp for all their fragments.

If a task is to go ahead with the execution, it requests the *Model Manager* to retrieve all required models in *loaded* condition. At this point the stamp of the input models is recorded by the *Execution Tracer*. After the execution of the task, the *Model Manager* computes the loaded hash of output models and the *Execution Tracer* records their new value. In the execution trace, a record of the latest output value is always kept to be able to detect external changes.

Context-awareness. After input and output models, and task parameters have been processed, if the task must go ahead with its execution then the *Model Manager* assigns to the task all model resources it requires for its execution, i.e. those to be consumed, modified and produced. This component creates and loads a model if none is available from previous executions or returns an instance of a model from a previous execution. The model types (e.g. EMF) decide if models that have been previously used can be re-used by the task. For example, a model that was previously loaded as an output model in another task could be configured by the EMF implementation to switch to read-only if is now being required as input.

After the task is executed and the traces produced by the task have been recorded by the *End-to-End Tracer* component, models that are not used by any future tasks are disposed.

3.5 End-to-end traceability

The execution of model management tasks may contribute a set of traces e.g. between model elements, or between elements and regions in text files. The *End-to-End Tracer* component is responsible for updating the model management trace model. ModelFlow's engine provides a generic interface, which model management task providers (e.g. ATL, Acceleo) can use to translate their traces into. This enables the component to keep all traces in the same format.

The end-to-end traceability metamodel is presented in Figure 5, and is an extended version of the one published in [13]. The root element is the *ManagementTrace* which contains a *TaskTrace* element for each task in the build. This element binds a collection of *Trace* elements to a task in the build. A trace supports arbitrary multiplicities of source and target *Elements* at various granularity levels. These elements are associated to a resource in the build which is their container. An element may be a *ModelElement*, a *ModelElementProperty*, a *File* or a collection of file *Regions*. Note that, in the case of model elements, they are expected to have unique identifiers within their model. The *Link* that connects source and target elements of a trace can have a *type* but can also specify the *name* of the *Operation* or rule associated to the trace e.g. a model-to-model transformation rule named *toleranceToComponent*. The metamodel also support attaching metadata to the traces by using *Property* elements.

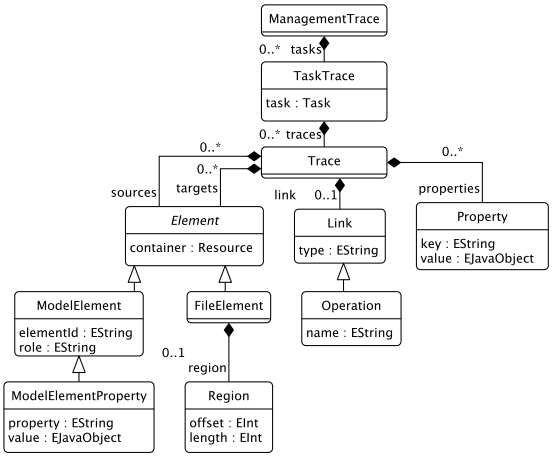


Figure 5: End-to-End Management Trace Metamodel

3.6 Implementation

Given the maturity of existing incremental build tools such as Gradle and the availability of relevant prototypes such as Pluto, the decision to build ModelFlow from first principles needs to be justified. At this stage, we have opted to implement ModelFlow as a standalone tool and not as an extension of an existing build system (e.g. Pluto/Gradle) to avoid any assumptions and technical constraints imposed by the architectures of these systems (e.g. Gradle does not support dynamic task outputs as discussed in Sec. 2.2). Having understood the information and mechanisms required to achieve conservative execution of model management tasks, context-aware

model loading and disposal, and end-to-end traceability, extending Gradle/Pluto to accommodate similar capabilities is mainly an engineering activity.

ModelFlow is currently implemented as a series of Eclipse Java plugins that extend the language and processing facilities of the Epsilon project. As a model-based project, it uses several metamodels to capture the workflow specification, the execution trace, and the end-to-end traceability. Currently, workflow specifications can be prescribed using a Java API or a concrete Epsilon-based syntax.

4 EVALUATION

We now present a qualitative evaluation of ModelFlow based on a common subset of features shared with Gradle.

4.1 Experiment setup

We have executed the workflow from the motivating example both in Gradle and ModelFlow under seven different scenarios. The first scenario represents a clean build, while all other scenarios represent realistic changes to resources (models, generated files) which affect subsequent executions. These scenarios are described in Sec. 4.2. Both build tools parse an equivalent build script that captures the workflow and uses the same model management tasks and resources.

Gradle setup. We have extended Gradle to support the execution of the EVL, ETL and EGX tasks required by the workflow. In addition, we have also extended its DSL to support a custom data structure where models can be defined once. The Gradle workflow specification is presented in Listing 5. Lines 1-14 illustrate a custom data structure that we implemented to capture the models. Each model indicates its type in brackets, while configuration parameters are captured within curly braces. The model management tasks of the workflow are declared in lines 15-33. Each task receives the names of its input and output models as parameters.

As a general-purpose build tool, Gradle does not support most of the desired MDE build tool features out of the box. Its conservative execution mechanism is based on inputs and *expected* outputs i.e. known before the task execution. In addition, dynamic resources such as models cannot influence the task execution order, there is no end-to-end traceability offered and outputs are not protected at any point.

Our Gradle task extensions for Epsilon have been implemented so that they resolve required input and output models from the model DSL extension and the model files are declared as dynamic inputs or outputs. We have some task parameters as inputs or outpus as we do in ModelFlow, however their hashes are computed with the default mechanism used by Gradle. Upon execution, our task implementations iterate over required input and output models, loading all required models before execution and disposing all after the execution.

```
1  epsilon {
2    models {
3      config(EMF){
4        modelFile = file('resources/m/config.model')
5        metamodelFile =
             file('resources/mm/configuration.ecore')
6      }
7      component(EMF){
8        modelFile = file('resources/m/component.model')
9        metamodelFile =
             file('resources/mm/component.ecore')
10     }
11     extended(EMF){
12       modelFile = file('resources/m/extended.model')
13       metamodelFile =
             file('resources/mm/component.ecore') }}}
14 task validate(type: EVL){
15   src = file('resources/mmt/validation.evl')
16   input = 'config'
17   input = 'component' }
18 task m2m(type: ETL){
19   src = file('resources/mmt/extended.etl')
20   input = 'config'
21   input = 'component'
22   output = 'extended'
23   dependsOn validate }
24 task m2t(type: EGX){
25   src = file('resources/mmt/generate.egx')
26   outputRoot = file('src-gen')
27   input = 'extended' }
```

Listing 5: Gradle workflow

ModelFlow setup. We ran ModelFlow in non-interactive mode (see Sec. 3.3) and configured it to discard any changes in the task outputs of tasks. No model management traces were recorded. These actions were taken to make ModelFlow's execution similar to Gradle's except from how up-to-date checks for task parameters and model resources.

4.2 Scenarios and Results

We describe below the set of changes that the different scenarios involved, along with the observed behaviour of the tools.

1) *Clean execution:* This scenario represents a first-time execution where no caches are available. Both tools behaved as expected, that is, all tasks were executed.

2) *No changes:* After a clean execution, in this scenario we trigger a new one having made no changes to input or output resources. As such, we wouldn't expect any task to be executed, which is the case for both tools in the experiment.

3) *Change in the source model:* In this scenario the *component* model file is modified after a clean execution by changing the name of a port in the *component* model. We expect everything to re-execute as *component* is a input model for the validation and model-to-model transformation tasks, and this property should be propagated to the extended model and into the generated code. In the experiment, this is the case in both tools.

4) *Change in intermediate output model:* In this scenario we modify the value of a filter element in the extended model after a clean execution. Using the non protective execution mode of ModelFlow, we expect it to trigger the transformation to restore the consistency of this model and to skip the code generation. A similar behaviour is expected from Gradle. This is the observed behaviour on both.

5) *Template changes:* After a clean execution, this scenario consists of triggering an execution after modifying the template files required by the model-to-text transformation. As this is the only task affected, we expect both tools to only execute that task. This is the observed behaviour on both build tools.

6) *Non-protected changes in generated code:* In this scenario, we add a comment outside of the protected regions of a generated file. In contrast to the previous scenario, it is the task's outputs that are modified not its inputs. In this case ModelFlow only executed the model-to-text transformation, overwriting the not-allowed changes in the generated code, while Gradle skipped all tasks, leaving the changes in the generated code.

7) *Protected changes in generated code:* In this scenario, we add a print statement inside a protected region of a file from the generated code. We expect all tasks to be skipped as the output should be considered up-to-date for the code generating task. In this case, both build tools behave as expected.

4.2.1 Performance analysis. We report on the execution times of the scenarios in which both tools reacted to changes in the same way. The time measure of their execution are shown in Figure 6. The value reported for the first scenario correspond to the time of the first execution (clean), while all other scenarios report on the time of the second execution.

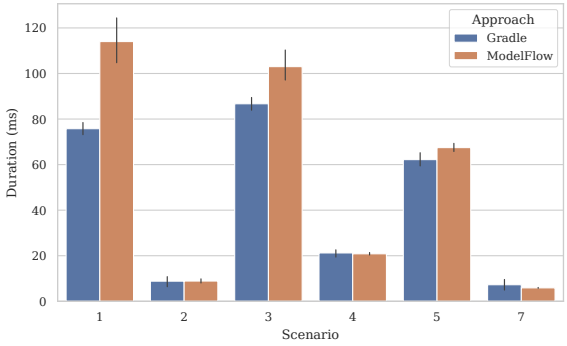


Figure 6: Execution time of each scenario in milliseconds.

The workflow was configured to use a *component* model (24kB) that represents a system of controllers (such as the one displayed in Figure 3a), and a *configuration* model (686 bytes) that was used to create a filter for each controller. Each scenario was executed 20 times with 5 warm-up iterations. We used Gradle version 6.2.1 and invoked it with the Gradle Tooling API ensuring no cache files were available between iterations. The experiments were executed on a 8-Core Intel Core i9 CPU @ 2.3 GHz with 16 GB of RAM and the Java Virtual Machine was provided with up to 4GB of memory running with JDK 1.8.0_231.

4.3 Discussion

While most scenarios resulted in similar behaviour on both built tools, we now discuss those that didn't.

In Scenario 4, ModelFlow can respond to changes in the *extended* model in two different ways: either to (a) use the modified model as source and trigger the code generation; or (b) discard the modifications in the model by triggering the transformation and skipping the code generation. However, this is not possible in Gradle which by default respond with the second approach which discards the changes invoking the transformation. Moreover, the reason why the model-to-model task is executed in Gradle is because the model file (declared as output) is known before the task execution.

In Scenario 6, ModelFlow executed the model-to-text transformation which was able to restore the build consistency while Gradle skipped the output analysis for the generated files which are known after the execution.

In Scenario 7 both tools behave as expected but for different reasons. ModelFlow does not execute because it determines that the outputs have not been modified from previous executions, while Gradle simply skips the output analysis for the same reasons as in Scenario 6.

Regarding performance, the computation of changes to input resources was slower in ModelFlow, particularly in the first-time execution. However, subsequent executions were nearly identical to Gradle's. The computation of output resources is more exhaustive in ModelFlow which may account for some of the overhead. While there was no mechanism to reuse loaded models in Gradle, the size of models used does no incur on a significant reloading overhead. In future evaluations, the use of larger models could highlight the impact that the mode reuse approach has on the workflow performance.

4.4 Threats to validity

One clear threat to validity are the implementation differences between the two build engines, inherent to their own architectures. We have minimised this threat by implementing the code of the invoked tasks as equivalent as possible so that the results of the evaluation reflect the impact of the architectural decisions and not of the individual tasks. Furthermore, we opted for a custom data structure to declare the models, so that tasks can receive the models in a similar fashion as tasks in ModelFlow do. The purpose of the performance evaluation was to give a time context to the qualitative evaluation but was not intended to measure the scalability of the approaches as the size of the models used in the experiment are small. That said, we have removed from the performance evaluation those scenarios in which the behaviour of the two tools was different.

5 RELATED WORK

There are several model management frameworks and tools that provide facilities to manage workflows such as the one presented in Sec. 2.1. We now discuss some of these tools with a focus on their incremental workflow mechanisms, context-awareness for model loading/disposal and end-to-end traceability facilities. As none of these tools have mechanisms to protect externally modified outputs, we leave this feature outside of the discussion.

MMINT is an extensible and graphical model management tool for exploration and experimentation [3] . At its core MMINT builds a megamodel that is described at two levels of abstraction: the *type-level* where metamodels are interrelated through relationships and megamodel operators (e.g. filter, map, reduce, merge) [3] defining the relationships and operations allowed for models at the *instance-level*. From their definition model management operations are strongly typed and to execute them they must be invoked manually and individually. In this context, only relevant model loading activities are triggered when editor views are opened and when

model management operations are invoked. Similarly, the strong and explicit typing of model relationships allows model management operations to produce trace links at model element level that become part of the megamodel. While the tool can be extended to support different metamodels, its scope seems to be limited to EMF-based models. To our knowledge, there is no dependency graph, task execution schedule nor execution incrementality as these features escape the rapid prototyping purpose of the tool.

MTC-Flow is a graphical tool that enables the definition and execution of chains of model management operations [1]. A workflow definition consists of the declaration of models and files that are consumed or produced by transformations (operations which use a model as input, output or both). These chains of operations are executed by identifying the input resources of the workflow and invoking the tasks that consume them. When a task finishes its execution it notifies that the output models and files are ready to be used by the tasks that use them as input. Before each operation is executed, validations may be performed on the models that it uses. In MTC-Flow, the workflow definition itself works as an explicit dependency graph. This tool supports a variety of model management tasks from different frameworks and its notion of a model is sufficiently abstract so that each task can implement their own model interpretation. That being said, for each task execution, models are created, loaded and disposed regardless of whether they are later reused by other tasks. Similarly, there are no validations to check whether an input or output file or model has changed from a previous execution to determine whether a re-execution is required. Regarding model management traceability, MTC-Flow does not seem to support it at any level.

MWE2 is a workflow engine that allows the definition of tasks that read/write EMF resources, perform operations on them and generate artefacts from them. It is worth noting that MWE2 is a language designed to be used by the Xtext language generator to configure itself. As such, it is not concerned with incrementality, managing execution or model management traces, nor with dealing with non-EMF models. The execution life cycle of MWE2 consists of three phases: pre-execution, execution and post-execution. At each of these phases, all tasks and sub-workflows invoke the method that corresponds to the phase in the order in which they are declared, i.e. sequentially. This execution process is therefore not engaged with task interdependencies. Regarding model handling, MWE2 relies on explicit tasks to read and write EMF models.

ChainTracker [5] is a state-of-the-art traceability tool that also supports the execution of model-to-model and model-to-text transformations using ATL and Acceleo, correspondingly. The main contributions of this tool are traceability collection and analysis in the form of visualisations. As such, we are not aware of any mechanisms in place for conservative workflow executions or context-aware model loading and disposal. Within its traceability model, ChainTracker not only considers model resources but also how metamodel constructs at attribute level are used by invoked rules in the model management tasks.

A summary of how the different tools support the desirable features listed in Sec. 2.1 is captured in Table 1. The "Conservative Execution" column indicates whether the tools have support for incrementality at workflow level. A full circle is given if they do out-of-the-box, half circle if they need to be explicitly specified outside the tasks and an empty circle if there is no known support. The "Context-Aware Resources" column indicates whether the tool can perform context-aware model loading and disposal. A full circle is given if models are loaded and disposed based on their use on the workflow. Half a circle is given if no model loading or disposal tasks need to be specified but models are loaded and disposed by the tool on every task they are used. An empty circle is given if the loading and disposal tasks must be invoked by the user. The "End-to-End Traceability" column indicates whether the tool provides model management traces. A full circle is given if end-to-end traces are provided as a by-product of the execution. Half a circle is given if individual task traces may be available if requested but not as part of an end-to-end product. An empty circle is given if no model management traces are available.

Tool	Conservative Execution	Context-Aware Resources	End-to-End Traceability
MTC-Flow	○	◐	○
MMINT	○	◐	●
MWE	○	○	○
ChainTracker	○	◐	●
ModelFlow	●	●	●

Table 1: Model management tools

6 CONCLUSIONS AND FUTURE WORK

We presented ModelFlow, a prototype MDE build tool designed to conservatively execute workflows that include model management tasks while offering end-to-end traceability and context-aware model loading. Our decision to implement ModelFlow from scratch was evaluated by attempting to replicate its behaviour with Gradle in a typical model-driven engineering scenario. Our evaluation validated this decision by highlighting scenarios that Gradle was unable to handle.

Future Work. We recognise that MDE tasks are only a subset of all tasks in a build and that attempting to develop a complete replacement for a build tool such as Gradle would be an ambitious task. Thus, in future work we will explore how to either propose extensions to Gradle to accommodate the scenarios we have identified or to provide integration mechanisms between Gradle and ModelFlow. Furthermore, we will investigate how ModelFlow copes with more complex workflows and concurrent execution.

ACKNOWLEDGEMENTS

This work was partially supported by Innovate UK and the UK aerospace industry through the HICLASS project (Contract #113213), the EPSRC in partnership with Rolls-Royce (Grant EP/R512230/1), and the NSERC via the Discovery Grant program.

REFERENCES

[1] Camilo Alvarez and Rubby Casallas. 2013. MTC Flow: A tool to design, develop and deploy model transformation chains. In *Proceedings of the workshop on ACadeMics Tooling with Eclipse - ACME '13*. ACM Press, New York, New York, USA, 1–9. https://doi.org/10.1145/2491279.2491286

[2] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. 2014. Neo4EMF, A scalable persistence layer for EMF models. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-09195-2_15

[3] Alessio Di Sandro, Rick Salay, Michalis Famelis, Sahar Kokaly, and Marsha Chechik. 2015. MMINT: A graphical tool for interactive model management. In *Proceedings of Model Driven Engineering Languages and Systems (MODELS)*. 82–97.

[4] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 89–106. https://doi.org/10.1145/2858965.2814316

[5] Victor Guana and Eleni Stroulia. 2014. ChainTracker, a model-transformation trace analysis tool for code-generation environments. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8568 LNCS (2014), 146–153. https://doi.org/10.1007/978-3-319-08789-4_11

[6] Victor Guana and Eleni Stroulia. 2019. *End-to-end model-transformation comprehension through fine-grained traceability information*. Vol. 18. Springer Berlin Heidelberg. 1305–1344 pages. https://doi.org/10.1007/s10270-017-0602-0

[7] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. 2008. Scalability: The holy grail of model driven engineering. *ChaMDE 2008 Workshop* (2008), 10–14.

[8] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. 2008. The epsilon transformation language. *Lecture Notes in Computer Science* 5063 LNCS (2008), 46–60. https://doi.org/10.1007/978-3-540-69927-9_4

[9] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. 2009. On the evolution of OCL for capturing structural constraints in modelling languages. *Lecture Notes in Computer Science* 5115 LNCS (2009), 204–218. https://doi.org/10.1007/978-3-642-11447-2_13

[10] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), 76–86. https://doi.org/10.1145/3238147.3238196

[11] Benjamin Muschko. 2014. *Gradle in Action* (1st ed.). Manning Publications Co.

[12] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and F. A C Polack. 2008. The Epsilon Generation Language. In *Model Driven Architecture – Foundations and Applications*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-69100-6_1

[13] Beatriz Sanchez, Dimitris S. Kolovos, and Richard Paige. 2019. Modelflow: Towards reactive model management workflows. In *DSM 2019 - Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling, co-located with SPLASH 2019*. https://doi.org/10.1145/3358501.3361238

[14] Beatriz Sanchez, Athanasios Zolotas, Horacio Hoyos Rodriguez, Dimitris Kolovos, and Richard Paige. 2019. On-the-Fly Translation and Execution of OCL-Like Queries on Simulink Models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 205–215. https://doi.org/10.1109/MODELS.2019.000-1

[15] The Eclipse Foundation. [n.d.]. CDO Model Repository. https://www.eclipse.org/cdo/