ModelFlow: Towards Reactive Model Management Workflows

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

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

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

Abstract

In this paper we propose a domain specific language that enables the description and execution of model management workflows. Our language declares tasks and resources involved in a multi-step model management process and resolves the execution behaviour and order based on dependencies among these components. We describe the abstract and a concrete syntax of the language along with its execution semantics. Then, we demonstrate how the language interpreter can orchestrate and execute a selection model management tasks through a case study of a workflow that generates a graphical editor from a metamodel.

Keywords Model Driven Engineering, Epsilon, Model Management, Workflow, Reactive

ACM Reference Format:
Beatriz A. Sanchez, Dimitris S. Kolovos, and Richard F. Paige. 2019. ModelFlow: Towards Reactive Model Management Workflows. In *Proceedings of DSM '19: Domain Specific Modeling Workshop (DSM '19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

It is through model management operations that concrete software development artefacts can be produced in an automated fashion from models [12]. However, due to the heterogeneity of models and model management languages, orchestrating them is an ongoing research topic [4, 15].

We propose a domain specific language that enables the execution of model management workflows from resource (e.g. File, EMF model) and task (e.g. model-to-model/text transformation, model validation) declarations. At runtime, an interpreter is able to resolve explicit and implicit dependencies among these components and propose a corresponding execution plan. The kinds of dependencies that are taken into consideration include: inter-resource references, task input and output resources, and inter-task dependencies. These dependencies can be explicitly declared but some can be inferred from the type of task or resource. The execution plan can either schedule all tasks or respond to a resource update and schedule only the tasks required to restore the overall consistency.

The rest of the paper is structured as follows. Section 2 discusses the motivation for this language. Section 3 presents the abstract and concrete syntax of the language along with its execution semantics and integration plans. Section 4 demonstrate how the language can capture a pre-existing model management workflow introduced in the motivation. Section 5 discusses related work. Finally, section 6 concludes the paper and presents future work.

2 Motivation

As a motivating example for this work, the first part of this section describes the automated graphical editor generation process of the EuGENia [8] tool which builds on the EMF and GMF code generation processes. Then, the section discusses why the existing implementation of EuGENia is not as efficient as it could be. The last part of the section describes envisioned features for a reactive model management workflow language which is later described in section 3.

2.1 EuGENia

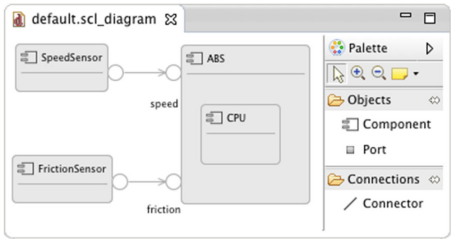
EuGENia is an existing open-source tool that predates this research and which uses metamodel annotations and model transformations to streamline the process of generating graphical model editors based on EMF and GMF [8]. For example, Listing 1 shows a metamodel defined in Emfatic (a textual notation for Ecore) which describes a Simple Component-connector Language (SCL) [8]. This metamodel has been extended with `@emf` and `@gmf` annotations placed on top of the various metamodel constructs in order to specify aspects of the EMF and GMF generation processes. The result of EuGENia's execution on this metamodel is the graphical editor in Figure 1a.

```
1  @namespace(uri="scl", prefix="scl")
2  @emf.gen(basePackage="org.eclipse.epsilon.eugenia.
       examples")
3  package scl;
4  @gmf.diagram
5  @gmf.node(label="name", color="232,232,232")
6  class Component {
7    attr String name;
8    @emf.gen(propertyMultiline="true")
9    attr String description;
10   @gmf.compartment(layout="free")
11   val Component[*] subcomponents;
12   @gmf.affixed
```

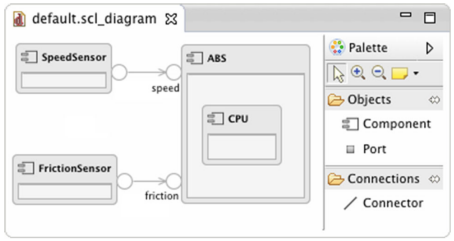
DSM '19, October 20–25, 2019, Athens, Greece
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
13    val Port[*] ports;
14  }
15  @gmf.link(source="from",target="to",label="name",
        target.decoration="arrow")
16  class Connector { 19
17    attr String name;
18    ref Port#outgoing from;
19    ref Port#incoming to;
20  }
21  @gmf.node(figure="ellipse",size="15,15",label.icon="
        false",label.placement="external",label="name")
22  class Port {
23    attr String name;
24    val Connector#from outgoing;
25    ref Connector#to incoming;
26  }
```

Listing 1. Annotated Emfatic metamodel of an SCL



(a) Initial



(b) Polished

Figure 1. SCL editor generated with EuGENia [8].

To enable the generation of a graphical editor from a metamodel, EuGENia extends and integrates the built-in EMF and GMF code generation processes described below.

EMF. The EMF code generation process starts from the definition of the domain metamodel (abstract syntax) in `Ecore` or an `Emfatic` [17] file. Then built-in EMF model-to-model transformations are used to produce the EMF generator model (`GenModel`) from the metamodel. The `GenModel` captures Java implementation details and can be further customised. Finally, an EMF built-in model-to-text transformation consumes the `GenModel` and produces the Java code and required configuration files.

GMF. The Graphical Modeling Framework (GMF) provides a model-driven approach to the generation of Eclipse-based graphical editors for EMF-based DSLs. Its code generation process builds on the EMF code generation process. The first stage of this process involves the manual construction of models that specify different aspects of the graphical syntax of the language. These models include the graph model (`GmfGraph`) which specifies the shapes, connections, labels, decorations, etc.; the tooling model (`GmfTool`) which specifies element creation tools; and the mapping model (`GmfMap`) which maps the graphical elements in the `GmfGraph` model with the creation tools of the `GmfTool` model and the abstract syntax elements of the Ecore metamodel. The second stage of the process involves the production of a generator model (`GmfGen`) from the mapping model. The generator model contains the implementation details required by the graphical editor code generator and is produced from a model-to-model transformation. In the last stage, code is generated from the generator model.

Why EuGENia? Without EuGENia, simple metamodel changes can be propagated to the corresponding `GenModel` by an EMF built-in reconciler without overwriting any user-defined customisations. However, for more complex changes the `GenModel` would need to be regenerated and customised from scratch [8]. EuGENia provides a set of built-in metamodel annotations to attach implementation semantics which can be used to customise the `GenModel` after it is generated. Some of these (starting with `@emf`) are illustrated in Listing 1.

Similarly, while GMF provides built-in wizards for generating the `GmfTool`, `GmfGraph`, and `GmfMapping` models from the metamodel, the resulting models are very simple [8]. Consequently, these models need to be manually created and updated after any metamodel changes [8]. With EuGENia, another set of built-in metamodel annotations can be used to attach graphical semantics to metamodel elements and enable the automated derivation of these models from the metamodel. Some of these (starting with `@gmf`) are illustrated in Listing 1.

Overall, after model-to-model transformation tasks, EuGENia triggers built-in fixes derived from the used annotations but also allows the execution of polishing transformations, that is, user-defined in-place model transformations that can be used to fine-tune the models produced by predefined model-to-model transformations and model fixes. After fixing and polishing, the initial SCL editor from the previous example could look like Figure 1b. In addition, EuGENia provides and executes built-in metamodel validations that, upon failure, halt the execution of subsequent steps of the EMF and GMF processes.

The inter-model references and model equivalences used in the overall graphical editor generation process are captured in Figure 2.

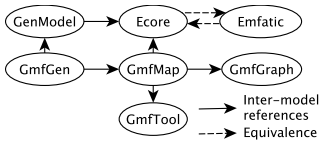


Figure 2. Inter-resource dependencies

2.2 Discussion

Consider the dependency graph in Figure 3 which shows all the steps (groups of tasks sharing a colour) executed

by EuGENia and the model resources they consume and produce.

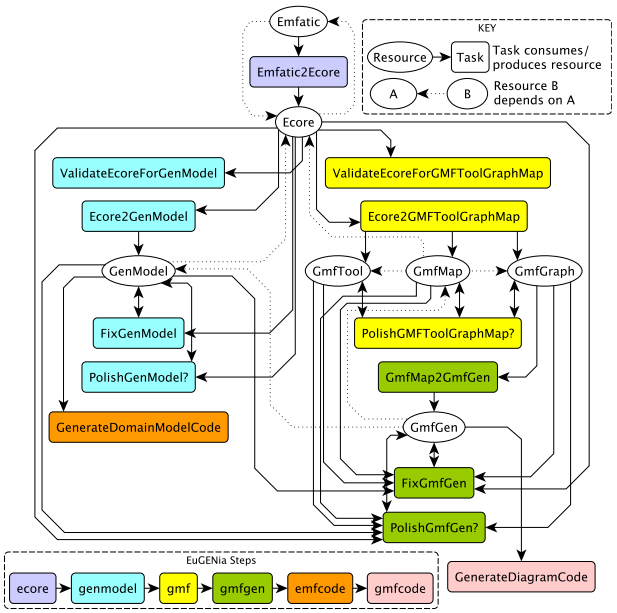


Figure 3. Eugenia task-resource and inter-task dependencies. The tasks in each step are sequentially ordered top-to-bottom. The solid arrows show the input and output models of the different tasks. Dotted arrows represent resource dependencies from Figure 2. Task names ending in a question mark denote optional tasks.

Concurrency. EuGENia's full workflow consists on the sequential execution of the steps `ecore`, `genmodel`, `gmf`, `gmfgen`, `emfcode`, and `gmfcode`. While this tool provides a much more efficient approach than plain EMF and GMF, we can tell from Figure 3 that some of those steps could be concurrently executed.

Reactiveness and Incrementality. Consider a situation in which a user removes the `@gmf.compartment` annotation in line 10 of the `Emfatic` file of Listing 1. The `Ecore` model resulting from `Emfatic2Ecore` would be different and so would the output models of `Ecore2GMFToolGraphMap`. However, the `GenModel` produced by `Ecore2GenModel` would remain the same as there was no change in for the EMF generation process. In this case, there would be no need to re-execute tasks such as `FixGenModel`, `PolishGenModel` or `GenerateDomainModelCode`. While EuGENia supports partial executions through the selection of sequential steps, ideally it should support incremental executions where the minimal number of tasks required in response to resource changes are scheduled. In addition, it would be convenient if EuGENia could react to resource changes and plan or even trigger execution of tasks dependent on the changes.

Change protection. Consider a scenario in which a user manually modifies the `Ecore` metamodel and re-executes EuGENia process from this change, that is, all task except `Emfatic2Ecore`. If the user later modifies `Emfatic`, the execution of `Emfatic2Ecore` would have the potential to overwrite the previous changes in `Ecore`. In this case, the user should be given the option to either discard the previous changes e.g. by triggering `Emfatic2Ecore`, or to block tasks that may overwrite the previous changes altogether. It would be convenient if EuGENia could detect this situations with the potential to overwrite external resource changes and either prompt before proceeding or accept parameters that instruct how to respond to them.

Smart loading. In EuGENia, required models are loaded before the execution of each task, and they are disposed when the task is finished. The large models can make more expensive the loading and disposing operations, even more when this process is involves the same model in several tasks. To make this process more efficient models should be loaded just before the execution of the task that first uses them, and disposed just after the task that uses them last is finished.

2.3 Features

The motivation for this work can be drawn from the previous discussion. Overall, we envision a framework that enables the declarative specification of tasks and resources, and can detect and react to resource changes based on declared dependencies among the model management tasks and resources. The envisioned framework should be able to support smart model loading and disposal based on a given model management execution plan. In addition, this framework should be able to prevent an execution when it has the potential to overwrite external changes in resources. Moreover, the envisioned framework should be able to compute consistency restoration plans and execute them when restricted resources are externally modified, for example, when generated code that is not meant to be tweaked is externally modified.

To achieve this we need a language and a supporting execution engine that can capture inter-resource, inter-task and resource-task dependencies from which execution plans can be drawn for first-time executions and executions driven by resource changes. The language should provide the ability to declare optional tasks, alternative execution paths, and to attach conditions on the execution of tasks possibly from the results of former task executions. Such a language is introduced in section 3.

3 The ModelFlow Language

In the following we present the abstract and concrete syntax of ModelFlow along with its execution semantics. The purpose of this language is to enable the declarative definition and reactive execution of workflows involving models and model management tasks. We provide rationale on why current tools such as Gradle or BPMN are not a good fit to support desired features of the framework in section 5.

3.1 Abstract Syntax

The abstract syntax of the language is captured in the metamodel presented in Figure 4. The metamodel constructs are described below.

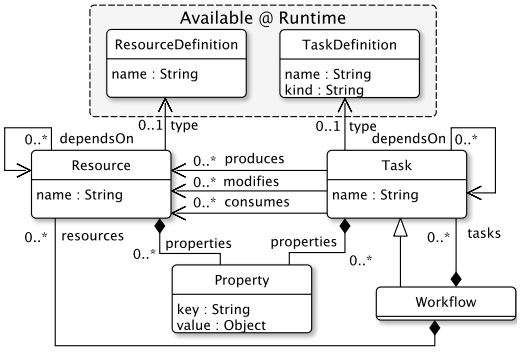


Figure 4. Workflow Specification Metamodel

ResourceDefinition. This named element refers to a concrete procedure to represent and manipulate a model resource that is attached to an underlying technology such as EMF, Simulink, File. These elements can check whether resource instances that reference them as type are valid.

Resource. A resource is any software artefact that can be manipulated in the workflow. The kind of resource is determined by its *type* which references a `ResourceDefinition`. The *properties* of the resource are used to configure the `ResourceDefinition`. For example, a resource named *GenModel* of type *EMF* could be configured with the property *src* with a value pointing to the EMF model file location. The `ResourceDefinition` should validate whether the `src` property is known and if its value is valid.

TaskDefinition. This named element defines the behaviour of a model management activity. The *kind* of activity is used as a classification of the model management activity such as validation, model-to-model transformation, etc. In contrast, the name of this element refers to a concrete model management implementation. For example a validation model management task that uses the Epsilon Validation Language (EVL) [11] is named `EVL` and classified as of `validation` kind. These elements can check whether task instances that reference them as type are valid.

Task. This named element represents a concrete executable model management activity. The *type* of model management activity to be executed is defined by the `TaskDefinition`. The *properties* of the task are used to configure the `TaskDefinition`. A task may reference *resources* that are to be consumed, modified or produced as a result of its execution. For example, a task named *myValidation* of type *EVL* could be configured with the property *src* whose value points to the location of the validation script and an input resource (consumes reference) that points to the resource *myModel* defined in the example above.

Workflow. A workflow element is the root of the model and it represents a composite activity, that is, an executable subprocess. Each workflow contains many *Resource* and *Task* elements. When started, a workflow gives control to its internal tasks and upon completion, the control is returned to the parent workflow.

Property. This element represents a key value pair used to configure tasks and resources.

3.2 Concrete Syntax

We now present the concrete syntax of the language. The language is mostly declarative to place the focus on the definition of tasks and resources and their inter-dependencies, and not on the order that the interpreter will execute them. The language also contains imperative fragments specified in the form of EOL [9] statements (e.g. in the guard) to support complex guards. Note that `ResourceDefinition` and `TaskDefinition` elements are not part of the concrete syntax as they are provided by the interpreter.

3.2.1 Resources

Based on the *Resource* construct described in the previous section, Listing 2 provides the concrete syntax for a resource definition. The resource declaration must start with either the `resource` or `model` reserved words to differentiate resources which need loading and disposing (i.e. the models). Then follow the name of the resource (`<Name>`) and the name of the *ResourceDefinition* type (`<Type>`). Optionally a list of comma separated resources (`<Resources>`) can be inserted after the `references` reserved word to declare dependencies. Inside the keys is contained a list of relevant properties for the resource type. The annotations on the resource declaration influence the resolution of the execution plan and their semantics will be described in the next section.

```
1  (@non-blocking)? (@root|@derived)?
2  (resource|model) <Name> type <Type>
3    (references <Resources>)?
4  {
5    (<Property.Key> (:expression|{statementBlock}))*
6  }
```

Listing 2. Concrete syntax of a resource declaration.

Listing 3 is an example of two resource declarations: the first for an *EMF* model named *GenModel*, and the second for a *FileSet* resource named *Sources*.

```
1  model GenModel type EMF {
2    src : "workflow.genmodel"
3    metamodel : "http://.../emf/2002/GenModel"
4  }
5  resource Sources type FileSet {
6    root { return "output"; }
7    includes : "**/*.java"
8  }
```

Listing 3. Resource declaration example.

3.2.2 Tasks

The concrete syntax of the *Task* construct is specified in Listing 4. A modeling task starts with the task keyword followed by the name of the task (`<Name>`) and its *TypeDefinition* type (`<Type>`). Optionally, the task may specify dependencies to other tasks which must be executed before it. Similarly, the task may specify a comma separated list of tasks (`<OnFailTask>`) to be executed upon the task's failure.

In addition to the list of property key value pairs as in the Resource declaration, inside the brackets a task may declare guards (statements that must evaluate to true in order to execute the task) and input (<in>), output (<out>) and in/out (<inout>) resources which correspond to the *consumes*, *produces* and *modifies* references in Figure 4. The concrete syntax of each of these resources (<TaskResource>) is defined (Listing 5).

```
task <Name> type <Type>
  (, dependsOn <Tasks>)? (, onFail <OnFailTask>)?
{
  (guard (:expression)|({statementBlock}))?
  (in(?)? : <TaskResource>(, <TaskResource>)*)?
  (inout(?)? : <TaskResource>(, <TaskResource>)*)?
  (out(?)? : <TaskResource>(, <TaskResource>)*)?
  (<Property.Key>(?)?(:expression|{statementBlock}))*
}
```

Listing 4. Concrete syntax of a task declaration.

```
((<taskPropertyName> as )?<Resource.Name>(?)?({
  (<Property.Key>(?)?(:expression|{statementBlock}))*
})?
```

Listing 5. Concrete syntax of <TaskResource>.

An example of a model validation task using the EVL language is presented in Listing 6. This task takes as input two models: the GenModel declared in Listing 3 and an EcoreModel resource whose *expand* property is set to true. Note that this task would only execute if the source (src) file exists as indicated by the question mark.

```
task validateGenModel type EVL {
  in : GenModel, EcoreModel {
    expand : true
  }
  out : self.validationTrace as TraceResource
  src? : "mmop/validate.evl"
}
@derived model TraceResource type JavaModel;
```

Listing 6. Task declaration example.

3.2.3 Workflow

Listing 7 provides the concrete syntax for a workflow construct definition. The workflow definition requires the workflow reserved word and a name for the workflow. It may declare parameters that may be used in its guard or by its contained resource and task declarations (<resourceDeclaration> and <taskDeclaration>) to configure their properties.

```
(@(primary|lazy)?
workflow <Name> {
  (param <Param.Name>(?)? : <Param.Type> (= <Default.
      Value>)?;)*
  (guard (:expression|{statementBlock}))?
  (<resourceDeclaration>)*
  (<taskDeclaration>)*
}
```

Listing 7. Concrete syntax of a workflow declaration.

Listing 8 provides an example of a workflow definition (Eugenia) followed by its invocation as a task (graphical-Editor) in the Main workflow.

```
@lazy workflow Eugenia {
  param sourceName : String;
  guard : sourceName.startsWith("Emf")
  task A type X {...}
}
@primary workflow Main {
  task graphicalEditor type Eugenia {
    sourceName : "EmfMetamodel"
  }
}
```

Listing 8. Workflow declaration example.

3.3 Execution semantics

This section provides a discussion of the language execution semantics. Section 3.3.1 discusses the dependency resolution mechanism. Section 3.3.2 describes how resources are monitored in order to trigger workflow executions. Then, section 3.3.3 illustrates how the dependency graph is used to build an execution plan. Finally, section 3.3.4 presents the execution trace model used to support incremental executions.

3.3.1 Dependency resolution

The task- and resource-type definitions such as EVL and EMF, respectively, are made available through the interpreter. While some definitions may be available by default, others could be registered at runtime. These task- and resource-type definitions are used to check if task and resource declarations in the workflow are valid and well-formed.

The next step after validation is the dependency resolution which produces a directed graph with interconnected task and resource nodes. The first step of this process consists on the insertion of declared resources and tasks as nodes. Then, directed edges among tasks are inserted based on declared explicit dependencies such as the dependsOn clauses. At the same time edges between resources and tasks are inserted based on the task's declared input, output and in/out resources.

Task- and resource- type definitions can also implicitly create dependencies with non-declared resources. Take for example the validation task in Listing 6 where the src attribute represents a validation script that the EVL task-type definition requires. A change in the script would imply that the validation task needs re-executing and consequently this resource is registered as an input of the task. Implicit dependencies are also inserted in the dependency graph.

3.3.2 Resource change event processing

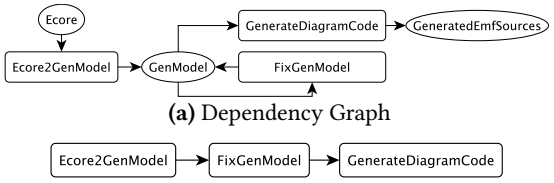
After a clean execution of the workflow, the ModelFlow engine starts monitoring all resources present in the dependency graph for changes. Each resource-type definition should provide a mechanism that can compute a unique *stamp* [15] for the resource that determines whether the resource has changed e.g. the last modification time of the resource or a hash of its contents. The monitor collects resource change events and re-computes their stamp. Changes derived from the workflow execution are excluded. If the value of the stamp changes, the resource is considered dirty and may

trigger an incremental workflow re-execution. Depending on the workflow's execution mode, resource changes may immediately trigger the execution or be collected until an execution is requested.

3.3.3 Execution plan

Once the dependency graph is constructed it can be used to assemble the execution plan. This is achieved by transforming the directed dependency graph (DDG) into a Directed Acyclic Task Graph (DATG).

The transformation identifies root resource and tasks and creates inter-task dependencies based on different criteria. For example, explicit task inter-dependencies are created immediately. Then for each resource, tasks that produce it go before tasks that modify it while tasks that alter it go before tasks that only read it. This situation is illustrated in Figure 5 where Figure 5a shows a dependency graph with resources `Ecore`, `GenModel` and `GeneratedEmfSources` and tasks `Ecore2Gen-Model`, `FixGenModel` and `GenerateDiagramCode` and its respective DATG in Figure 5b which resolves `Ecore` as root resource and makes `Ecore2GenModel` (produces `GenModel`) go before `FixGenModel` (modifies `GenModel`) and `FixGenModel` go before `GenerateDiagramCode` (reads `GenModel`).



(a) Dependency Graph

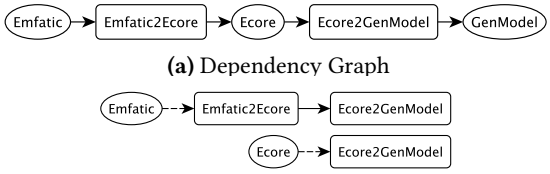
(b) Directed Acyclic Graph

Figure 5. Example with task-resource dependencies

Once the DATG is built, it can be iterated in topological order, triggering the execution of each visited task. When a task node has multiple outgoing edges, the outgoing paths can be executed independently (and therefore, potentially concurrently). In contrast, if a task has multiple incoming edges it must wait for the incoming paths to complete.

In a clean execution of the workflow, the full DDG is transformed into a DATG. In contrast, when the workflow must be executed from a specific task or when it is triggered by a resource (or group of resources) update, then the DATG is produced from a sub-graph of the DDG which consists of dependencies relevant for the starting task or, respectively, for the changed resources.

Consider the dependency graph in Figure 6a. A change in resource `Emfatic` or `Ecore` would respectively trigger the executions in Figure 6b. The change in `Emfatic` would trigger an execution equivalent to a *clean execution* but a change in `Ecore` would result in a partial execution where `Ecore` and `GenModel` would be consistent with each other as they are derived from the change in `Ecore` but `Emfatic` would be left out. In this scenario, a later change in `Emfatic` would potentially overwrite `Ecore` as it would trigger `Emfatic2Ecore`. To protect the changes in `Ecore`, ModelFlow would by default prevent the execution of any tasks that have the potential to overwrite it. Alternatively, `Ecore` could be marked in a way that does not block any tasks from overwriting its external changes, therefore enabling `Emfatic` to execute. This is the role of the `@non-blocking` annotation in a resource declaration.

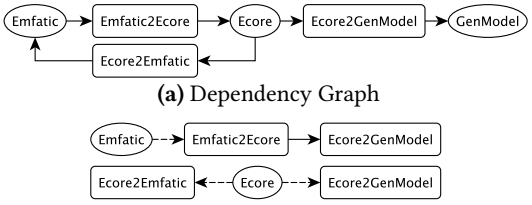


(a) Dependency Graph

(b) Directed Acyclic Graph

Figure 6. Complete/partial execution based on resource changes

Consider the dependency graph in Figure 7. In this case, the graph shows a circular dependency between resources `Ecore` and `Emfatic` through the `Emfatic2Ecore` and `Ecore2Emfatic` tasks. In absence of a `@root` annotation on either `Ecore` or `Emfatic`, one would be randomly selected as root in a clean execution. Both execution plans are illustrated in Figure 7b which are equivalent to execution plans based on changes on these resources. Despite not all tasks being executed when `Ecore` or `Emfatic` are selected as roots, all consumed/produced resources are up-to-date as a result of the execution, therefore there is no need to block any task from executing to protect changes in either `Ecore` or `Emfatic`. In contrast, if `GenModel` was externally modified, `Ecore2GenModel` executions resulting from modifications of `Ecore` or `Emfatic` would be blocked.



(a) Dependency Graph

(b) Directed Acyclic Graph

Figure 7. Complete execution based on resource changes

Before an execution plan is carried out, ModelFlow would check that the plan has no impact on any resources protected from overwriting. Before the execution of each task, ModelFlow checks if all inputs (explicit and implicit) are up-to-date comparing their stamps against those in the execution trace. If all inputs are up-to-date the execution of the task is skipped.

For a group of tasks in which a resource is produced or modified, the resource is managed in a transactional way so that if the output of the first task is equivalent to that of the task's previous execution then the resource is rolled back to the version of the previous execution. This is only valid if all other inputs in remaining tasks in the group remain the same, otherwise the task are executed. For example in Figure 5b

where `Ecore2GenModel` and `FixGenModel` produce and modify `GenModel`, this task group would roll-back to the previous version of `GenModel` if the outcome of `Ecore2GenModel` was the same as in its previous execution since `FixGenModel` only has `GenModel` as input, that is, it can be skipped. The rationale for this is that if ModelFlow skips `FixGenModel` for having the same resources as in last execution but does not roll-back `GenModel` then this resource would not go through the modifications of `FixGenModel` and would therefore be left in an unfinished state. Alternatively, if the resource-type definition does not support rolling-back procedures then the members in the task group would all be executed so that all modifications on the resource are applied.

Rules. Below are some rules on the establishment of resource-task dependencies (consumed, produced, modified) which are used to build the execution plan and help to break dependency cycles.

- For each resource, only one task should produce it as output. If several, their execution must be exclusive.
- If several tasks modify the same resource, their execution must be made sequential. If no explicit dependency among these tasks is found, their order would be chosen randomly.
- There is no restriction on the number of tasks that can read a resource.

3.3.4 Execution trace model

At a clean execution, the trace model is created. The `ExecutionTrace` element is the root of the model and points to the main workflow file. For the clean execution and any subsequent ones, a new `WorkflowExecution` element will be created in the root. These elements contain the set of triggering resources (if any) as `ResourceSnapshot` elements, in addition to the list of tasks that were executed as `TaskExecution` elements. Each `ResourceSnapshot` points to a workflow `Resource` and records its stamp at a given time. `TaskExecution` elements point to a declared `Task` and refer to a list of input `ResourceSnapshot` elements and contain new output `ResourceSnapshot` elements. In addition, an `TaskExecution` element keeps a list of its predecessors and successors. The `ExecutionTrace` root keeps a list of the latest `ResourceSnapshots` for each declared `Resource` and also a list of currently protected `Resources`, that is, those with potential to be overwritten by other tasks.

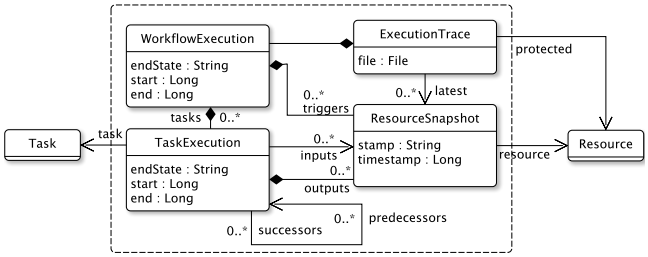


Figure 8. Workflow Execution Trace Metamodel

With this configuration it is possible to identify whether resources have changed from previous executions and if tasks can overwrite changes on protected resources.

3.4 Integration with build systems

We envision the framework's ability to (a) execute non-modeling tasks and (b) be executed from popular task execution engines such as Maven, Gradle or Ant.

To enable the execution of Maven, Gradle or Ant tasks from ModelFlow, we envision their declaration as regular tasks that do not accept any declared input, output or in/out resources but which may implicitly register resources such as the script (`src`) file as inputs and any produced resources as outputs. Furthermore, these task could specify the set of tasks to be executed in the form of goals for Maven, tasks for Gradle and targets for Ant.

```
1  task compileWithMaven type Maven {
2    src : "dir/pom.xml"
3    goal : "compile:compile"
4  }
5  task compileWithGradle type Gradle {
6    src : "dir/build.gradle"
7    task : "compile"
8  }
9  task compileWithAnt type Ant {
10   src : "dir/build.xml"
11   target : "main"
12 }
```

To enable the execution of ModelFlow from build tools like Maven, Gradle or Ant, a plug-in can be registered in their task repositories where full or partial workflow invocations can be made. The workflow could register the execution trace in a local repository to enable incremental executions aware of resource changes.

```
1  <model.flow src="myfile.mflow"/>
```

4 Case study: EuGENia

In this section we demonstrate ModelFlow's capacity to capture a real model management scenario through a case study that defines a reactive workflow that can generate a graphical editor based on the EuGENia (section 2). Note that while the EuGENia tool only uses EMF models, the language can support other modeling formats such as Simulink.

The workflow presented in Listing 9 defines the resources, tasks and configuration parameters that EuGENia requires to produce the graphical editor. The first line is the declaration of the workflow which requires the `modelName` parameter (line 2) to be set in order to locate all required models. Optionally, the `polishBasedir` parameter (line 3) can be set up to locate polishing scripts.

This workflow declares 9 resources: one `File` resource, 2 `FileSet` resources and 6 `EMF` models. `EMF` models must specify the model file (`src`) and the metamodel (`metamodel`) they conform to. The `GeneratedEmfSources` and `GeneratedGmfSources` resources are derived.

This workflow declares 14 tasks. Tasks of types ETL, EOL and EVL are transformation, query and validation tasks implemented using Epsilon's respective languages [10, 10, 11]. The previous task types require a script (`src`) which is an implicit input resource. The `Emfatic2Ecore` task is a text-to-model transformation that transforms an Emfatic file into an Ecore model. `EcoreCodeGenerator` and `DiagramCodeGenerator` are model-to-text transformations that take an model of EMF type as input and generate code. The generated output files can be accessed through the task's `generatedFiles` property and which are assigned to the `GeneratedEmfSources` and `GeneratedGmfSources` resources by `EcoreCodeGenerator` and `DiagramCodeGenerator` respectively. Finally, `GmfTransformToGenModel` is a built-in GMF model-to-model transformation which produces the `GmfGen` model.

```
workflow EuGENia {
param modelName : String;
param polishBasedir? : String = baseDir;
pre {
  var basefile : String = baseDir + modelName;
}
resource Emfatic type File {
  src : basefile + ".emf"
}
@non-blocking model Ecore type EMF {
  src : basefile + ".ecore"
  metamodel : "http://.../emf/2002/Ecore"
}
model GenModel type EMF {
  src : basefile + ".genmodel"
  metamodel : "http://.../emf/2002/GenModel"
}
model GmfGen type EMF {
  src : basefile + ".gmfgen"
  metamodel : "http://.../gmf/2009/GenModel"
}
model GmfMap type EMF {
  src : basefile + ".gmfmap"
  metamodel : "http://.../gmf/2008/mappings"
}
model GmfTool type EMF {
  src : basefile + ".gmftool"
  metamodel : "http://.../gmf/2005/ToolDefinition"
}
model GmfGraph type EMF {
  src : basefile + ".gmfgraph"
  metamodel : "http://.../gmf/2006/GraphicalDef..."
}
@derived resource GeneratedEmfSources type FileSet;
@derived resource GeneratedGmfSources type FileSet;
task Emfatic2Ecore type Emfatic2Ecore {
  in : Emfatic
  out : Ecore
}
task ValidateEcoreForGenModel type EVL {
  in : Ecore
  src : "mmop/evl/Ecore2GenModel.evl"
}
task Ecore2GenModel type ETL dependsOn
    ValidateEcoreForGenModel {
  in : Ecore
  out : GenModel
  src : "mmop/etl/Ecore2GenModel.etl"
}
task FixGenModel type EOL {
  in : Ecore
  inout : GenModel
  src : "mmop/eol/FixGenModel.eol"
}
task PolishGenModel type EOL dependsOn FixGenModel {
  in : Ecore
  inout : GenModel
  src? : polishBasedir + "FixGenModel.eol"
}
task GenerateDomainModelCode type EcoreCodeGenerator {
  in : GenModel
  out : self.generatedFiles as GeneratedGmfSources
}
task ValidateEcoreForGMFToolGraphMap type EVL {
  in : Ecore
  src : "mmop/evl/ECore2GMF.evl"
}
task Ecore2GMFToolGraphMap type EOL dependsOn
    ValidateEcoreForGMFToolGraphMap {
  in : Ecore
  out : GmfMap, GmfGraph, GmfTool
  src : "mmop/eol/ECore2GMF.eol"
}
task PolishGMFToolGraphMap type EOL {
  in : Ecore
  inout : GmfMap, GmfGraph, GmfTool
  src? : polishBasedir + "ECore2GMF.eol"
}
task GmfMap2GmfGen type GmfTransformToGenModel {
  in : Ecore, GmfMap, GenModel
  out : GmfGen
}
task FixGmfGen type EOL {
  in : Ecore, GenModel, GmfGraph, GmfTool, GmfMap
  inout : GmfGen
  src : "mmop/eol/FixGMFGen.eol"
}
task PolishGmfGen type EOL dependsOn FixGmfGen {
  in : Ecore, GenModel, GmfGraph, GmfTool, GmfMap
  inout : GmfGen
  src? : polishBasedir + "FixGMFGen.eol"
}
task GenerateDiagramCode type DiagramCodeGenerator {
  in : GmfGen
  out : self.generatedFiles as GeneratedGmfSources
}}// close task and workflow
```

Listing 9. EuGENia workflow definition

The previous workflow produces the directed dependency graph shown in Figure 9.

4.1 Scenarios

In this section we introduce a series of scenarios based on the previous workflow definition and we describe the expected execution plan, as the engine has not yet been implemented. The scenarios are accompanied by figures that illustrate execution plans as sequence of tasks. Some resources may be present to highlight their relation with tasks as a input/output (solid arrow) or as a workflow trigger (dashed arrow).

4.1.1 Clean execution

Consider the first-time execution of the workflow. Because most tasks use `Ecore` as input and only `Emfatic2Ecore` produces `Ecore` the execution plan algorithm can infer that `Emfatic` is the root of the execution plan and `Emfatic2Ecore` the first task to be executed. From the full DDG, the resulting execution plan (DATG) is presented in Figure 10. The

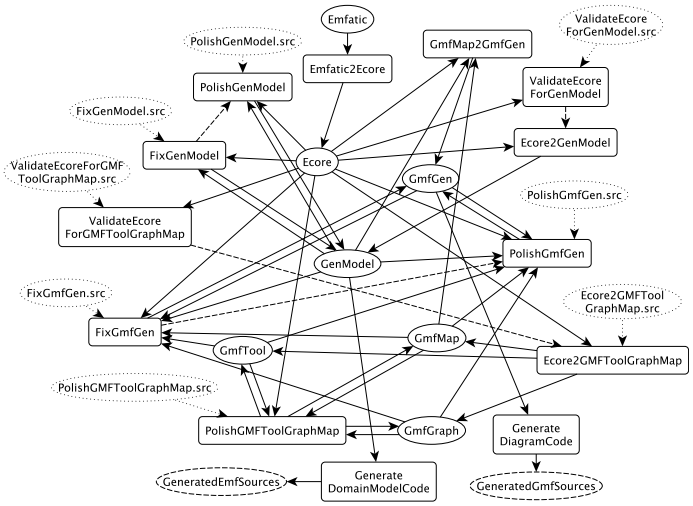


Figure 9. Simplified directed dependency graph. Tasks are denoted by rectangles and resources by ellipses. Dotted ellipses represent implicit resource declarations (such as a script file). Dashed ellipses represent derived resources. Solid arrows represent resource-task dependencies, dashed ones inter-task dependencies and dotted ones implicit resource-task dependencies. Inter-resource dependencies (Figure 2) and additional implicit resources are left out for simplicity.

figure shows two independent paths that can be executed concurrently after `Emfatic2Ecore`. However, due to the dependency of `GmfMap2GmfGen` to resources produced in both branches, both paths need to be finished before proceeding with this task. While the execution of `PolishGenModel`, `PolishGMFToolGraphMap` and `PolishGmfGen` would be scheduled, they would only be executed if their source file exists.

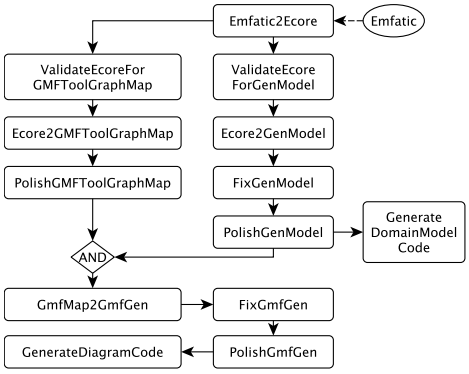


Figure 10. Workflow's first execution sequence. Similar execution plan when `Emfatic` is modified.

4.1.2 Resource modifications

`Emfatic`. A change in this resource produces a plan equivalent to a clean execution. If the change in the `Emfatic` resource produced a different `Ecore` resource as a result of `Emfatic2Ecore`, both `ValidateEcoreForGMFToolGraphMap` and `ValidateEcoreForGenModel` would be triggered and since `Ecore2GenModel` and `Ecore2GMFToolGraphMap` depend on them, they would also be executed. Alternatively if the produced `Ecore` resource did not change there would be no need to roll-back to a previous version, as there are no tasks that modify it, nor to continue with the workflow, as everything would be up-to-date.

If the `Ecore2GenModel` task produced a different `GenModel` than in last execution, then `FixGenModel` and `PolishGenModel` would need to be executed. In contrast, if the `Ecore2GenModel` task produced the same `GenModel` as in its previous execution, this resource would have to be rolled-back as there are two other tasks that modify this resource and stopping the execution after `Ecore2GenModel` would leave the resource in an inconsistent state.

`Ecore`. Consider an execution triggered by the modification of the `Ecore` resource. A similar execution plan as in Figure 10 would be triggered except that the first task would no longer be `Emfatic2Ecore` but instead the two concurrent execution paths that start with `ValidateEcoreForGMFToolGraphMap` and `ValidateEcoreForGenModel`, respectively.
This is because the workflow that reacts to a resource change starts where the resource is read and not where it is produced or modified. Since the `Emfatic` resource is not involved in any of tasks of this execution plan, at the end of the execution it would not be up-to-date with all other resources. However, since the `Ecore` model is annotated with `@non-blocking`, any modifications to the `Emfatic` would trigger `Emfatic2Ecore` and its subsequent tasks discarding any previous external changes in the `Ecore`.

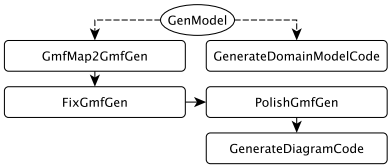


Figure 11. Execution plan when `GenModel` is modified.

`GenModel`. Consider an execution triggered by the external modification of the `GenModel` resource. In this case, Figure 11 would be executed and the tasks `Ecore2GenModel`, `FixGenModel` and `PolishGenModel` would be blocked.

`GeneratedEmfSources`. Consider a change in one of the output files produced by the `GenerateDomainModelCode` model-to-text transformation, that is, the `GeneratedEmfSources` resource. In this case any subsequent execution plan which involved the execution of the `GenerateDomainModelCode` task would have it blocked to protect the external changes of this resource.

Similar execution plans can be produced when `GmfTool`, `GmfGraph`, `GmfMap`, `GmfGen` are modified. For tasks that use configuration parameters as input resources (such as the source file of the task), upon their modification we can assume that the workflow would be triggered with these tasks as starting points.

5 Related Work

General purpose business process and workflow modeling languages such as BPMN [13], YAWL [19], UML Activities [5]

can be used to define model management processes. However, these languages are focused on the specification of the *control flow* instead of the dependencies among tasks and resources. The idea behind resource-task dependencies is that different execution plans are possible based on resource update events. The orchestration of model management activities can also be achieve with graphical model management tools such as MMINT [4], MTC-Flow[2] and Wires* [14]. Like the aforementioned modeling languages, these tools also place emphasis on the execution flow and while some specify consumption/production relationships with models, they do not use them to determine an execution plan. Other textual model management orchestration tools such as MWE2 [18], MoScript[7] only support sequential definition and execution of activities.

Build tools such as Ant [16], Gradle [3], and Pluto [6] have made progress regarding incremental execution plans. These tools have the ability to check for resource changes and react accordingly but don't protect modified resources from being overwritten by default. In addition, these tools rely their incrementality on file-based resource changes. Our language is intended to support multiple criteria based on the resource type definition. For example, a database model such as PTC-Integrity Modeler [1] should be able to detect modifications based on information from the database. Moreover, Ant's relies on "excessive use of [the] `uptodate`" [6] command to support incrementality.

6 Conclusions and Future Work

This paper presented a domain specific language that enables the reactive execution of model management tasks by capturing dependencies among task and resources. We presented the concrete and abstract syntax of the language along with its execution semantics and integration facilities with other build tools such as Maven, Gradle and Ant. In particular, the execution semantics make use the task and resource inter-dependencies to propose execution plans that may be blocked to protect externally modified resources from being overwritten or be triggered to discard external manual modifications. We demonstrated how the proposed language can capture a model management scenario that is based on the model management activities performed by the EuGENia tool which produces graphical editors from annotated metamodels.

Future Work. We are in the process of implementing the execution engine for this language, along with basic model management tasks and resource types. In the future we plan to add support for its integrations with other build tools and to add tooling support which includes an editor and graphical views of the workflow dependencies and execution plans.

Acknowledgement

This work was partially supported by the Engineering and Physical Sciences Research Council (EPSRC) through the National Productivity Investment Fund (NPIF) in partnership with Rolls-Royce (EP/R512230/1), and by the Mexican National Council for Science and Technology (CONACyT) (602430/472773).

References

[1] [n.d.]. PTC Integrity Modeler. http://www.ptc.com/en/products/plm/plm-products/integrity-modeler

[2] 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.

[3] Adam L. Davis and Adam L. Davis. 2016. Gradle. In *Learning Groovy*.

[4] 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.

[5] Marlon Dumas and Arthur H.M. ter Hofstede. 2001. UML Activity Diagrams as a Workflow Specification Language. *Proceedings of the UML'2001 Conference* (2001), 76–90.

[6] 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.

[7] Wolfgang Kling, Frédéric Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. 2012. MoScript: A DSL for Querying and Manipulating Model Repositories. In *SLE 2011: Software Language Engineering*. 180–200.

[8] Dimitrios S. Kolovos, Antonio García-Domínguez, Louis M. Rose, and Richard F. Paige. 2017. Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software and Systems Modeling* 16, 1 (2017), 229–255.

[9] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. 2006. The Epsilon Object Language (EOL). *Lecture Notes in Computer Science* 4066 LNCS (2006), 128–142.

[10] 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.

[11] 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.

[12] Dimitrios S. Kolovos, Massimo Tisi, Jordi Cabot, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, and Dániel Varró. 2013. A research roadmap towards achieving scalability in model driven engineering. *Proceedings of the Workshop on Scalability in Model Driven Engineering - BigMDE '13* (2013), 1–10.

[13] O M G Document Number and Associated Schema Files. 2012. *Business Process Model and Notation*. Lecture Notes in Business Information Processing, Vol. 125. Springer Berlin Heidelberg, Berlin, Heidelberg.

[14] José E Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. 2009. Orchestrating ATL Model Transformations. In *Proc. of MtATL 2009*. 34–46.

[15] Perdita Stevens. 2018. Towards sound, optimal, and flexible building from megamodels. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems - MODELS '18*. ACM Press, New York, New York, USA, 301–311.

[16] James Sugrue. 2015. Apache Ant. *DZone* (2015).

[17] The Eclipse Foundation. [n.d.]. Emfatic. https://www.eclipse.org/emfatic/

[18] The Eclipse Foundation. [n.d.]. MWE 2 - XText. https://www.eclipse.org/Xtext/documentation/306_mwe2.html

[19] W.M.P. van der Aalst and A.H.M. ter Hofstede. 2005. YAWL: yet another workflow language. *Information Systems* 30, 4 (jun 2005), 245–275.