# The Cupid Integrated Development Environment for Earth System Models

Rocky Dunlap[1]
College of Computing
Georgia Tech
rocky@cc.gatech.edu

April 3, 2014


[1]This work supported by the NASA CMAC program.

# Contents

# Chapter 1

# Overview of Features

Cupid is a set of development tools to facilitate the adoption of the geoscience modeling frameworks into new and existing model codebases. The target framework is the Earth System Modeling Framework (ESMF) and its interoperability layer called the National Unified Operation Prediction Capability (NUOPC), which is currently being implemented in most major climate and weather models in the US. Cupid tools are intended for model developers who have prior experience with model development workflows, but are new to developing with ESMF and NUOPC. It is also aimed at developers interested in exploring the benefits of using the Eclipse Integrated Development Environment (IDE) for improving development productivity.

Use of modeling frameworks is quickly becoming the norm for both operational and research climate and weather models. Modeling frameworks provide a number of benefits including mechanisms for componentizing complex codebases, increased developer productivity through code reuse, improved quality and robustness of features compared with "home grown" solutions, and fast execution via parallel data transfer and interpolation operators.

In a framework-based application, some application behaviors are provided by the framework and some are provided by the application developer. The framework provides a set of abstractions, *framework-provided concepts*, that the developer is required to instantiate and configure in code. Creating a framework-based application is called *framework completion* because the developer fills in application behaviors not provided by the framework, or specializes existing behaviors.

The Cupid tools adds domain-specific intelligence to the Eclipse Integrated Development Environment in order to facilitate adoption of ESMF and NUOPC. The features include:

- A tool for **reverse engineering** an existing codebase to determine what ESMF and NUOPC framework concepts are present in the code. The reverse engineered model is presented to the user alongside the source code in the form of a tree where nodes correspond to framework concepts. Clicking on

a node takes the user directly to the relevant code fragments. The reverse engineering tool also checks for code-level compliance to NUOPC technical rules and offers suggestions for addressing compliance issues.

- A tool for **automatic source code generation** of NUOPC-compliant code fragments directly in new or existing source files. The generated code helps the developer see what framework calls are required, where they should be located in the code, and what parameters the developer must provide.

- A **cloud configuration** feature that allows the user to select a training scenario and, within a few minutes, configure, compile, execute, and view the output of both skeleton models and realistic models executing on virtual machine instances on the Amazon EC2 platform.

# Chapter 2

# ESMF and NUOPC

This section describes the Earth System Modeling Framework (ESMF) and the National Unified Operational Prediction Capability (NUOCP) and provides references for those interested in finding out more. Those readers already familiar with ESMF and NUOPC may choose to skip this section.

## 2.1 The Earth System Modeling Framework

ESMF is a high-performance software framework designed for numerical geoscience models. Some of the framework-provided concepts include model components (`ESMF_GridComp`) and coupler components (`ESMF_CplComp`; mediators between model components), and data types for model state (`ESMF_State`), distributed arrays (`ESMF_Array`), physical fields (`ESMF_Field`), and numerical grids (`ESMF_Grid`; discretization schemes). An ESMF-based application is typically designed as a hierarchy of model components where components communicate by exchanging `ESMF_State` objects via framework-provided interfaces. `ESMF_GridComp`s and `ESMF_CplComp`s have user-customizable `initialize()`, `run()`, and `finalize()` methods. For more information about ESMF, see the ESMF User's Guide and the ESMF Reference Manual.

## 2.2 The National Unified Operational Prediction Capability

To promote interoperability of model components, NUOPC is a set of generic components, metadata conventions, and behavioral protocols encoded in a software layer on top of ESMF. Together, these elements form the basis of a *common model architecture*—a standard way of building models in order to make it easier to assemble coupled models using components from different sources. NUOPC is currently being implemented in research and operational models such as the HYCOM ocean

model [**?**], GFDL's MOM5 ocean model [**?**], and NASA's ModelE climate model [**?**]. Additional information about NUOPC can be found on the NUOPC home page.

NUOPC applications are built by combining four basic building blocks called *generic components*. The four types of generic component are `Driver`, `Model`, `Mediator`, and `Connector`. Many component behaviors have been predefined by NUOPC. This both promotes code reuse and facilitates interoperability. However, in some cases, the developer needs to provide implementations of behaviors not defined by NUOPC. Additionally, if the generic behavior does not meet the requirements of the coupled model, the developer may need to override existing behaviors. In both cases, the developer's implementation is typically provided in subroutines which are registered with and called by the framework. The process of providing new behaviors or overriding existing ones is called *specialization*. As defined here, *specialization* is conceptually similar to how a class overrides a parent class method to provide a different implementation in an object-oriented programming language. However, because the public ESMF and NUOPC APIs are not implemented in an object-oriented language, a custom specialization mechanism has been defined. Understanding the specialization process is essential for adopting NUOPC into a model's codebase.

The generic component `Driver` implements a harness of ESMF components and `ESMF_State` objects and it is specialized by plugging in `Model`, `Mediator`, `Connector`, and other `Driver` components. The `Driver` initializes its child components according to an *Initialize Phase Definition* and drives their `run()` methods according to a *Run Sequence*. `Model` wraps a user's code so it can be plugged into a `Driver`. `Model`s represent major geophysical domains such as atmosphere, ocean, and ice. `Connector`s and `Mediator`s manage communication between `Model`s. `Connector`s implement simple connections such as parallel redistribution or regridding of fields and `Mediator`s implement complex `Model` interactions requiring customized code. Figure 2.1 illustrates several possible architectural configurations of NUOPC components.

To take full advantage of NUOPC, developers must ensure that model components comply with NUOPC architectural constraints and technical rules. The full definition of NUOPC compliance is available on the NUOPC compliance web page.

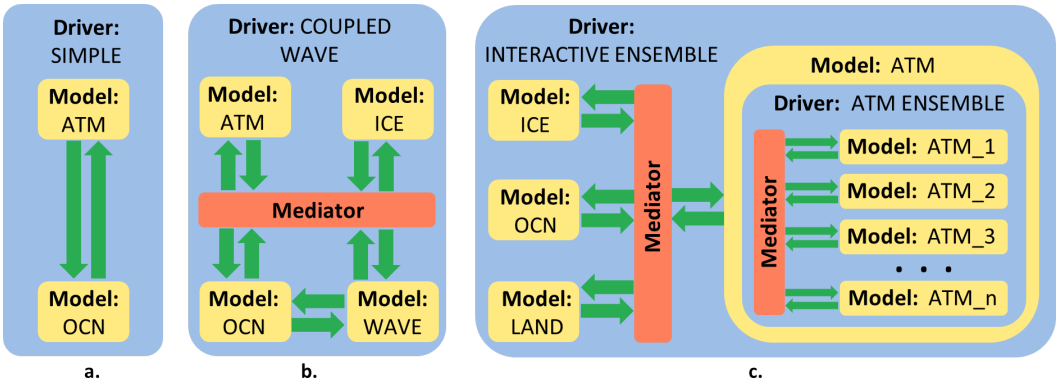


Figure 2.1: **a.** A `Driver` with two child Models and simple Connectors. **b.** A configuration in which a `Mediator` couples atmosphere, ocean, ice, and wave `Models`. **c.** A complex configuration showing nested `Drivers`.

# Chapter 3

# Reverse Engineering and Compliance Verification of NUOPC Applications

Most climate and weather model codebases are staggeringly large and obtaining an overview of the model, its subcomponents, and their interconnections is a cumbersome, time-consuming task. Often, this can only be accomplished by manually reading through the top-level source files to establish a mental picture of the overall structure of the model and its data flows.

Instead of viewing a model as an opaque, complex set of source files, Cupids reverse engineering feature parses a models source code and produces a high level representation of the framework concepts that are present. The reverse engineering feature is a kind of static analysis because the source code is analyzed before it is compiled and does not require execution of the model. This representation is shown to the user alongside the source code in an outline form called the NUOPC tree viewer. This provides an alternative, more abstract perspective for viewing a models source code. Some of the main concepts provided by NUOPC are Drivers, Models, Mediators, and Connectors. If a codebase contains any of these generic components, the reverse engineering function will automatically find them and present them in outline form. This provides an architectural overview of an entire coupled system without requiring the developer to read through thousands of lines of source code.

NUOPC ensures interoperability of modeling components by specifying a set of technical rules that model implementers should follow. In addition to presenting a high level view of framework concepts in source code, Cupids compliance checking feature provides feedback to the user when potential compliance issues are discovered in a reverse engineered model. For example, when developing a NUOPC Model component, certain subroutines must be implemented and registered with the framework. If a required subroutine or its registration is missing, Cupid can

identify the problem and annotate the outline view with icons indicating that some required code is missing. Moreover, this feedback is provided immediately to the user during model development thereby reducing the number of runtime failures and improving efficiency of the development process.

# Chapter 4

# Generating NUOPC-compliant Code

Even if a software framework is well designed, writing framework completion code is notoriously difficult, even for seasoned developers. Often, completing a single logical task such as adding a new run initialization phase or run phase to the model requires making several code additions at multiple places spread throughout the application source code. If one or more of the required additions are inadvertently left out, the application may not behave as expected. In the software engineering research community, many ideas have been proposed for how to help developers write framework-based applications correctly and efficiently. For ESMF and NUOPC, guidance is provided in the form of comprehensive API documentation (ESMF, NUOPC), system tests (included with source distribution), and small archetypical codebases that show how to structure NUOPC applications based on the components in the modeled system (e.g., standalone atmosphere, coupled atmosphere-ocean, three-component system, etc.).

Cupid's code generation feature complements these static resources by generating on-the-fly NUOPC-compliant source code fragments directly in existing source files. The user initiates a code fragment generation by adding elements to a reverse engineered model in the NUOPC tree viewer. The source code is then synchronized with the tree viewer, generating the required code fragments. The generated code fragments can then be customized by the developer for their particular case. The following use case illustrates use of forward engineering feature:

> A developer has finished writing the initialization phases for a NUOPC Model component called ATM and now needs to add the capability to advance the model one time step. The developer right clicks on the ATM element in the NUOPC tree viewer and selects "Add Model Advance." Two things happen immediately: the tree viewer is updated with a new sub-element underneath ATM called "Model Advance" which in turn

> contains sub-elements "Registered in Set Services" and "Implementation." Then, source code fragments are generated inside the Fortran file for ATM including a call inside the ATM SetServices to register the Model Advance subroutine and a stub for the new Model Advance subroutine.

The use case shows some advantages of this approach compared to an approach in which code is copied-and-pasted from archetypical example code. First, the new Model Advance element added to the tree viewer included multiple sub-elements indicating that source code changes are required in at least two places: a new subroutine and a call to register this subroutine with the framework. This provides guidance to the developer to ensure that all framework requirements are met. The approach, therefore, is less error-prone than brute force copy and paste. Also, the generated code fragments are customized based on the state of the existing source code. For example, the developer may be using specialized variable names. Since these variables have already been discovered during the reverse engineering phase, the generated code can reference these variables instead of requiring the developer to modify variables in copy-pasted code.

# Chapter 5

# Cloud-based configuration

Cupid simplifies the process of configuring a computational environment capable of compiling and executing high-performance geoscience models. IDEs package a lot of development tools into a single application to help manage and simplify the software development workflow. Although IDEs aim to increase developer productivity, they can still introduce a steep learning curve. Some challenges with using IDEs for geoscience model development include:

- Understanding the basic steps involved in moving from source code to a running model
- Making sense of the many development tools and features available in the IDE
- Setting up a high-performance computational environment capable of configuring, compiling and executing model code
- Configuring the IDE to connect to remote computational environments

Cupid's cloud integration feature allows a developer to select a training scenario and, within a few minutes, configure, compile, execute, and view the output of both skeleton models and realistic models. This feature relies on a set of pre-configured machine images that can be instantiated on Amazon EC2 cloud infrastructure. The machine images contain all of the necessary software dependencies for the selected scenario. Furthermore, Cupid automatically configures the IDE to connect to and synchronize source code with cloud-based virtual machines.

# Chapter 6

# The Behind-the-Scenes Meta-tool

This section describes theoretical and implementation aspects of the meta-tool used to define the mappings from framework-provided concepts to source code. This chapter may be of interest to software engineering researchers or those interested in using Cupid for defining their own framework-specific development tools. Knowledge of this chapter is not important for users who wish to use the tool to develop NUOPC-based applications.

To support model developers in writing NUOPC-complaint code, the Cupid tool leverages existing work aimed at facilitating development of framework-based applications called Framework-Specific Modeling Languages (FSMLs). A FSML is a domain-specific language designed for a specific framework [1]. FSMLs are aimed at addressing some of the challenges involved in developing framework-based applications, especially knowing how to complete a framework correctly and how to ensure respect of its rules of engagement[1]. The language elements in the FSML's abstract syntax represent framework-provided concepts that the developer instantiates in code. FSMLs can be represented as a *feature model* [**?**], where features correspond to framework-provided concepts and each feature model configuration represents a valid framework completion.

Bidirectional mappings from framework-provided concepts to application source code are used to support forward and reverse engineering functions. In the forward direction, framework completion code is generated from an FSML instance—i.e., a Framework-Specific Model (FSM). In the reverse direction, an existing codebase is analyzed in order to recognize code patterns that correspond to framework concepts, producing a FSM. Taken together, the forward and reverse mappings enable round-trip engineering: the developer can seamlessly move between two perspectives, a zoomed in code-level perspective for customizing source code, and a higher level perspective showing the framework concepts present in the application and their

inter-relationships.

## 6.1 The NUOPC Framework-Specific Modeling Language

This section describes a subset of the abstract syntax of the NUOPC FSML. Figure 6.1 shows some of the framework-provided concepts in the NUOPC FSML represented as features in a feature model. The top-level concept node is `NUOPCApplication`. Its child features, `NUOPCModel`, `NUOPCDriver`, and `NUOPCMediator`, are the primary architectural components defined by NUOPC that require specialization by the developer. Subfeatures of `NUOPCModel` include `genericImports`, a container feature for required and optional module imports, `implementsSetServices`, which represents a framework-called subroutine that the developer implements, `initialization`, whose (elided) subfeatures represent the initialization subroutines that the developer implements, and `implementsModelAdvance`, which represents the developer-provided subroutine that advances the model forward in time.

*Mapping definitions* define the correspondence between features in the FSML and structural and behavioral code patterns. Cupid defines a set of mapping types for Fortran 90, the primary language binding supported by ESMF and NUOPC. Currently, only structural mappings types are supported, and they have been defined on an as-needed basis. Mappings are indicated in the figure next to each feature. `NUOPCModel`, `NUOPCDriver`, and `NUOPCMediator` each have a mapping definition of `module`. The top-level concept, `NUOPCApplication`, does not indicate a mapping. It will be implicitly mapped to an entire codebase. (Think of this as the root directory of a source tree.) Table 6.1 lists the current set of supported mapping types.

The mapping definition for the feature `implementsSetServices` is `subroutine: "#name(inout type(ESMF_GridComp) #gcomp, out integer #rc)"`. A subroutine signature includes the subroutine name, and optionally the intents (in/out/inout), types, and names of its formal parameters. Only subroutines with a matching name and matching argument intents and types are matched. In general, mapping definitions may include meta-variables that refer to features. The subroutine signature above contains three meta-variables, `#name`, `#gcomp`, and `#rc`. The `#name` meta-variable implicitly maps the subfeature `name` to the name of the mapped subroutine.

In the figure, some features are marked with a `!` indicating that they are *essential* features. A feature is only instantiated if all of its essential child features can be successfully mapped. For example, in Figure 6.1, the features `importsGenericSS` and `callsGenericSetServices` are essential. If their mappings fail, then the higher-level feature `NUOPCModel` will not be be instantiated.

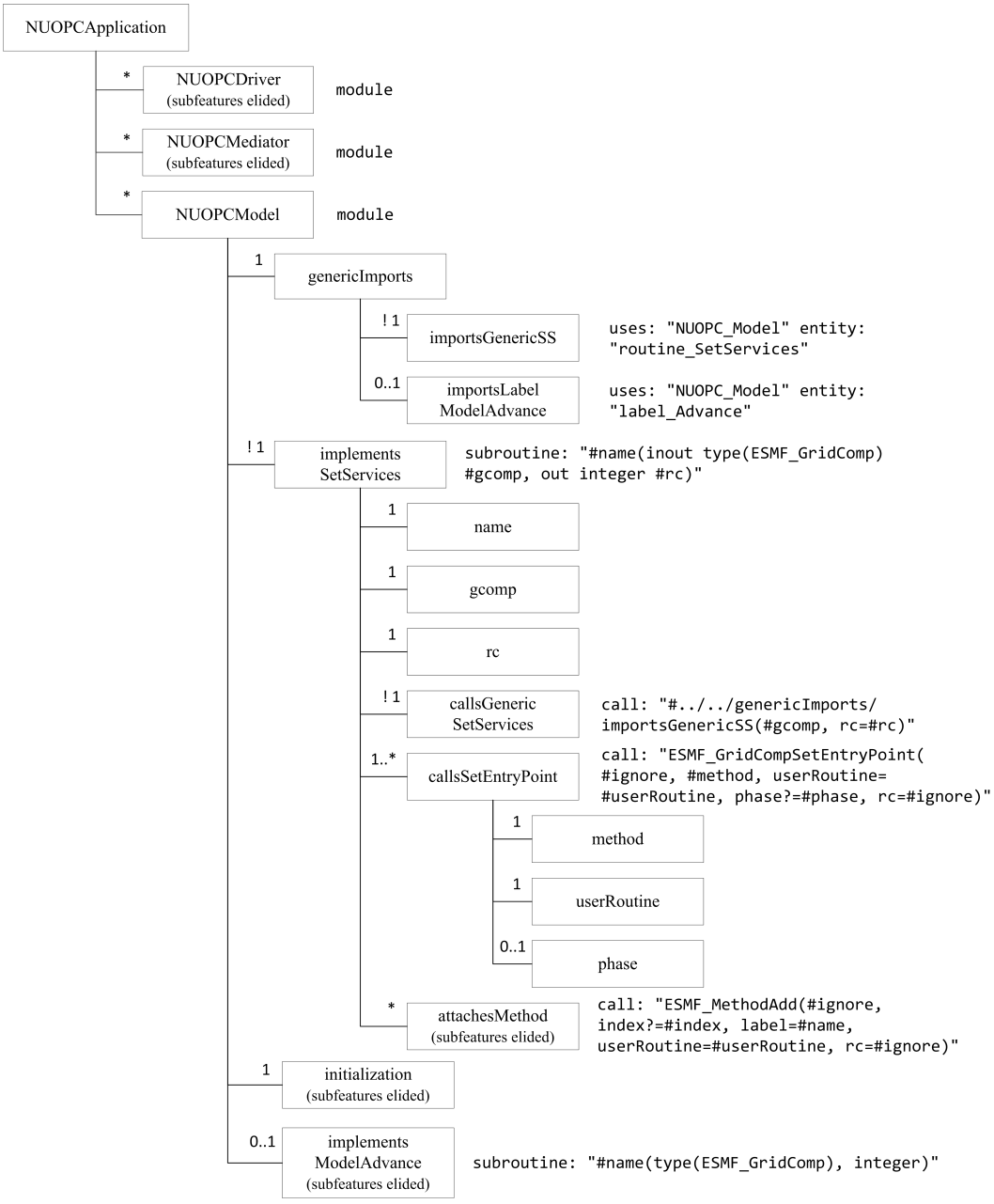


Figure 6.1: A partial feature model representation of the NUOPC FSML

Table 6.1: Mappings types for Fortran 90

| Structural Pattern Expression | Structural Elements Matched |
|---|---|
| `module` | matches a Fortran module |
| `m moduleName` | matches the name of module `m` |
| `m usesModule: mn` | matches use statements in module `m` with imported module name `mn` |
| `u usesEntity: en` | matches import of entity named `en` within use statement `u` |
| `m uses: mn entity: en` | matches use statements in module `m` with imported module name `mn` and imported entity name `en` |
| `m subroutine` | matches subroutines defined within module `m` |
| `m subroutine: ss` | matches subroutines defined within module `m` with signature `ss` |
| `s subroutineName` | matches the name of subroutine `s` |
| `s formalParam: i` | matches the `i`th formal parameter of subroutine `s` |
| `s call` | matches calls with the implementation of subroutine `s` |
| `s call: cs` | matches calls with the implementation of subroutine `s` with call signature `cs` |
| `c argByIndex: i` | matches the `i`th actual parameter of call `c` |
| `c argByKeyword: k` | matches the actual parameter with keyword `k` of call `c` |

## 6.2 FSML Implementation

In Cupid, a FSML is implemented as an Ecore model to take advantage of the Eclipse Modeling Framework (EMF) suite of tools. Ecore is an object-oriented meta-model and it includes a graphical Eclipse-based editor for creating class models and generating Java code. While classes represent framework-provided concepts, annotations on classes and their properties are used to specify mapping definitions. Figure 6.2 shows a UML class diagram for part of the NUOPC FSML rooted at the feature `NUOPCModel`. Features are represented as classes or attributes and subfeatures are represented as containment references. Figure 6.3 shows an instantiation of classes (a FSM) and mappings to source code.

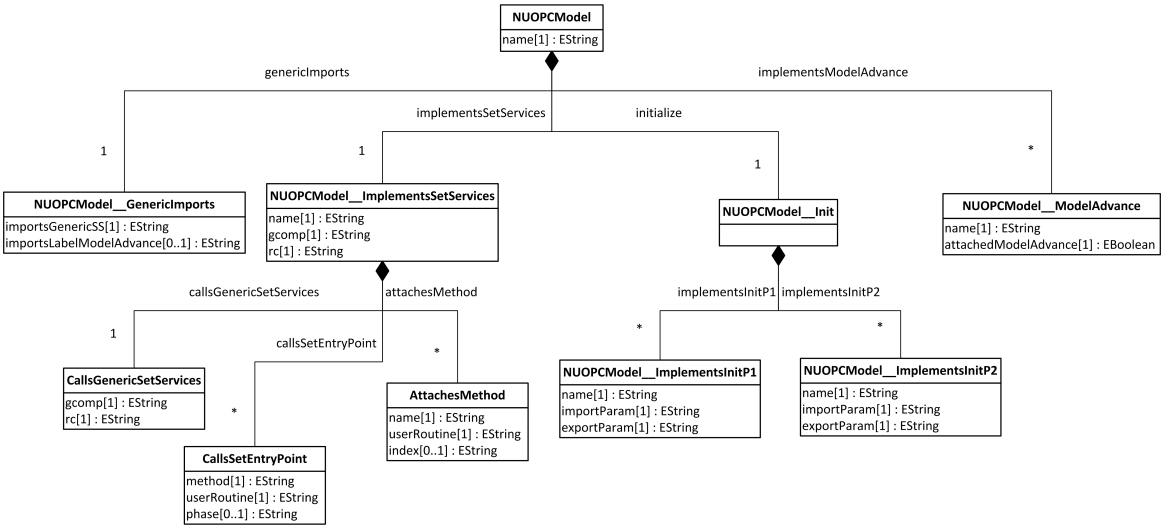


Figure 6.2: Partial NUOPC FSML class diagram rooted at `NUOPCModel`.

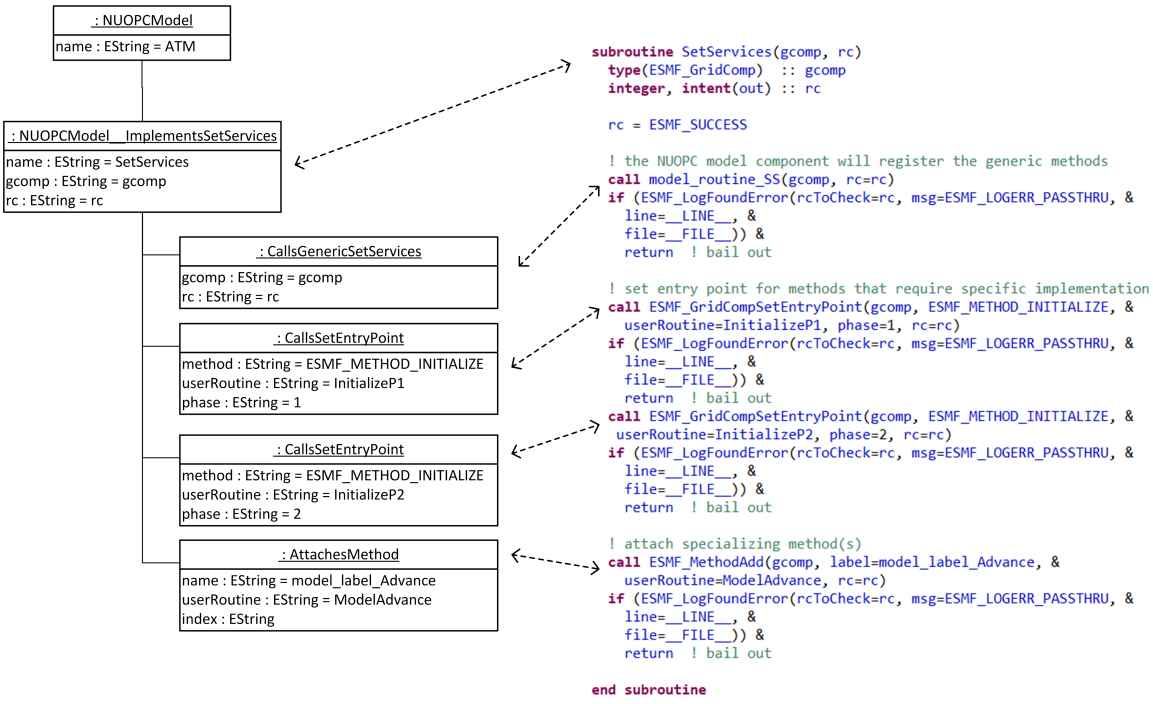


Figure 6.3: The object diagram on the left shows an instantiation of part of the NUOPC FSML corresponding to a `Model SetServices` subroutine. The dashed lines show mappings from FSM objects to source code.

# Bibliography

[1] Micha? Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 692–706. Springer Berlin Heidelberg, 2006.