Feature modelling: manageable system evolution through automatic generation

Sarah Colige

November 1, 2011
1 Introduction

2 Background

2.1 Community Earth System Model (CESM)

“The Community Earth System Model (CESM) is a fully-coupled, global climate model that provides state-of-the-art computer simulations of the Earth’s past, present, and future climate states.” [7]

The Community Earth System Model was created by the National Center for Atmospheric Research (NCAR) in 1983 as a freely available global atmosphere model for use by the climate research community. Its main purpose is to study climate variability and global change. Currently, CESM allows to include models of the atmosphere, land surface, ocean, and sea ice. These components are coupled with or without resorting to “flux adjustments” depending on the experimentation.

The most recent version CESM 1.0.3 was released on June 2011.

2.2 Community Atmosphere Model (CAM)

The Community Atmosphere Model (CAM) is the atmosphere model component of the CESM. The last version is 5.1. It is used both as a standalone model or as the atmospheric component of the CESM.

A standalone model usage means that the atmosphere is coupled to an active land model (CLM), a thermodynamic only sea ice model (CICE), and a data ocean model (DOCN). Researchers speak of “doing CAM simulations” in that case. When CAM is coupled to active ocean and sea ice models then they refer to the model as CESM. [6]

2.3 Users and benefit of CAM

CAM and, more generally, CESM are developed for the climate research community. Its primary aim is understanding and predicting the climate system.

“The long-term goals of the CESM are:

• to develop and to work continually to improve a comprehensive CESM that is at the forefront of international efforts in modelling the climate system, including the best possible component models coupled together in a balanced, harmonious modelling framework;
• to make the model readily available to, and usable by, the climate research community, and to actively engage the community in the ongoing process of model development;
• to use the CESM to address important scientific questions about the climate system, including global change and interdecadal and interannual variability; and
• to use appropriate versions of the CESM for calculations in support of national and international policy decisions.” [7]
3 Motivation

3.1 The CAM configure script

The configuration process of CAM\textsuperscript{1} is designed to support a number of different scientific scenarios. Because of this flexibility, configuration of CAM is a complex process. To run CAM in standalone mode, the user will execute the configure script and provide a number of command line parameters that describe his or her particular configuration choices. The user can only supply parameters that differ from the default values of most configuration options. The final compilation will include only certain source code folders based on the user’s choices.

One primary function of the CAM configure script is to enforce constraints among the configuration options. The configurable nature of CAM makes it is more like a Software Product Line (SPL) than a single software application. In other words, the CAM codebase can be used to create a large number of related software applications that vary in different ways.

Feature model diagrams offer a convenient and simple notation for documenting common and variation points in SPL. Feature diagrams also allow you to describe more or less simple constraints among the features. This should be helpful, at least, for visualizing the variation points (configuration options) of CAM and constraints among the different configuration options.\textsuperscript{3}

Thus currently, CAM uses a configure script in Perl to check constraints among the configuration options and select appropriate source directories to finally produce a Makefile and a Pathfile according to the parametrization. But this parametrization must be done by a domain expert familiar with the configure script of CAM, otherwise he or she could receive a series of error messages. Moreover, maintaining this kind of script has a significant risk of becoming inconsistent. The best simple example of inconsistency is conflicting constraints.

CAM configuration process The CAM standalone configuration process may be more explicit with the Figure 1 on page 4 that shows the current process step with a Roman numeration and the future process step with an Arabic numeration. This future process will be describe in the paragraph 5 on page 5. The current configuration process is:

I **Execute** The user executes the configure script with the list of parameters which represents the configuration.

II **Select** If the parameters meet all constraints and don’t raise an ERROR message, then the script selects all appropriate source directories.

III **Produce** And finally, the script produces a Makefile and a Pathfile. These files are used in the build phase.

\textsuperscript{1}Community Atmospheric Model
4 Feature modelling

4.1 Text-based Variability Language (TVL)

TVL is a text-based language for feature models with a C-like syntax. The goal of the language is to be scalable, by being concise and by offering mechanisms for modularity, and to be comprehensive.

Feature models are a common way to represent variability in software product line engineering (SPL). For this purpose, a graphical notation based on FODA is generally used. But those approaches have a obvious drawback that is their lack of scalability: they generally do not fit real-size problems. Indeed, their graphical syntax does not account for attributes or complex constraints and becomes a burden for large feature models.

A text-based feature modelling notation like TVL is both light and comprehensive, meaning that it covers most constructs of existing languages, including cardinality-based decomposition and feature attributes. The main objective of TVL is to provide engineers with a human-readable language supporting large-scale models through modularisation mechanisms. Furthermore, TVL can serve as an extensible storage format for feature modelling tools. [2]
However, TVL is currently a language proposal and feedback from the variability modelling research community is being requested. A number of case studies are in progress (this one is part of them). And a reference implementation for TVL in Java is also being developing, more details are available on the TVL website.

5 The future system

The primary goal of the future system is to automatically produce the inputs required by a configurator and corresponding to the input TVL model with a generator. These inputs will be composed of a translated model in the dimacs format with a feature-dimacs map, a translated model that can be used by a system with a graphical user interface and a feature-directories map. This configurator will manage a large part of the current configure script that means they will have the same semantic, while leaving the possibility to add custom code. The external GUI will make the configuration process easier than the current configure script, and produce a configuration file interpreted by the configurator.

Future configuration process & update management

The future process explained below shows only the basic usage of the configurator: (see Figure on page 4)

A Generator At first, inputs of the configurator must be generated with a TVL model.

B Configurator Then, users can create a configuration.

1 Execute The user executes the configurator and the GUI is displayed. He can creates a configuration.

2 Enforce constraints Each time a choice is made by the user, the constraints are checked by a solver and some options are auto-selected, disable or still “unknown”.

3 Run When there is no more “unknown” option, the configuration is valid and can be used to run the configurator.

4 Select Similarly than before, all appropriate source directories are selected.

5 Produce And finally a Makefile and a Filepath are produced. These files are used in the build phase.

5.1 Functionalities

The system will be composed of two subsystem: a generator and a configurator. The generator allows to generate the inputs of the configurator based on a TVL model. The configurator allows to create or edit a configuration, check a configuration and create a CAM standalone model. See the Figure on page 5

Generate configurator Users can generate the inputs of the CONFIGURATOR. They provide a TVL model to the GENERATOR and receive the required inputs of the CONFIGURATOR according to the provided TVL model.

http://www.info.fundp.ac.be/acs/tvl
5.2 Input/output

The generator receives a model encoded in TVL and produces a translated model in dimacs format with a feature-dimacs map, a feature-'human readable name' map, a feature-directories map and a translated model in a format supported by the GUI system. These outputs are produced only if the model is satisfiable. A GUI system using a SATsolver shows and enforces constraints expressed in the model to produce a configuration. While the configurator checks a configuration, selects the source directories according to the configuration and produces a Makefile and a Filepath that only contains paths for the CAM components. It uses the features map to link the feature id used in configurations from the GUI system to the feature name used in the TVL model. The configurator aims to replace the current configure script. See the Figure 3 on page 7.

---

Figure 2: Use case diagram of the system.

**Edit configuration** Users can edit a configuration with the CONFIGURATOR. A GUI is provided to make the process easier.

**Check configuration** Users can submit a configuration to the CONFIGURATOR to checks its validity.

**Create CAM standalone model** Users can create a CAM standalone model with the CONFIGURATOR by providing a configuration. If they use the GUI to edit a configuration the process is automatized.

These functionalities are explained in more details in the Annexe A on page 13.

---

3 Until TVL doesn’t support String attributes, this map must be handwritten, it’s represented by a dashed arrow on the the Figure 3 on page 7.

4 A model is satisfied if at least one configuration exists in which all constraints are met.
5.3 Component architecture

This subsection describes the system architecture.

The system is composed of two main components: a Generator and a Configurator; and few others components: a Visitor, a TemplatesManager, a ConfigurationTranslator and a ScriptExecutor.

- The Generator uses the Visitor to walk through the AST produced by the TVL-Parser to produce outputs described above in the Section 5.2 on page 6.

- The Configurator is composed by three subcomponents: an Editor, a Checker and a Builder, and uses a ConfigurationTranslator.

  - The Editor uses an API to communicate with a GUI System which uses a SATsolver and edit a configuration.
  
  - The Checker checks a configuration translated by the ConfigurationTranslator and uses the SATsolver.
  
  - The Builder uses the Checker and the ScriptExecutor to produce a Makefile and a Filepath.

- The Visitor provides different visitors to walk through the structure of a TVL model.
produced by the TVLParser and uses a TemplatesManager to format it in another language syntax.

- The ConfigurationTranslator translates different configuration formats to a format that can be used by the Configurator.

- The ScriptExecutor manages script executions. For example, it executes Perl scripts with the PerlProcessor.

See the Figure 4 on page 8.

Figure 4: Component diagram

5.4 The GUI

This section describes a sketch of the GUI. To make it more explicit, it’s based on the following hypothetical sample TVL model. See the Listing 1 on page 9. That could be represented by an incomplete graphical feature diagram made with FeatureIDE. See the Figure 5 on page 9.

Keep in mind that currently TVL doesn’t support String attributes.

Without attributes
5.4 The GUI

```java
struct myStructure { int num; String nom;}

root MyRoot
group allof{
    MyFeature
group oneof{
        Foo,
        Bar
    },
    opt MyFeature2
group allof{
        Baz
    }
}

MyFeature{
    int myInt in [1..4];
    real myReal in [0.5..1];
    enum myEnum in {One, Two, Three};
    bool myBool;
    myStructure myStruct {num is 253;}
    String myString is "tadaam";
}
```

Listing 1: myFeature TVL model

According to this sample, the main window allows users to parametrize a hierarchical configuration. This is inspired by the FeatureIDE configurator\[8\]. See the Figure 6 on page 10. Each feature is represented by its name in a tree hierarchy as they are defined and preceded by an icon what means the feature is selected, unselected or not yet parametrized. On the top, users are informed of the validity of the current configuration and the number of remaining possibilities.

Furthermore, in TVL, each feature can have attributes what can be parametrized in a popup window similar to the Figure 7 on page 10. On this figure, from top to bottom, there are a notification area, a parametrization area and an action area. The notification area provides users with different kind of informative messages. The parametrization area allows users to see all attributes and edit the value of the enabled attributes.

Due to the limited time to develop the system, the GUI will not be developed at first and an online service named SPLOT\[5\] will be used to figure the main window of the configurator.

---

According to constraints in the TVL model, some attributes have to be unchanged
6 Tools

6.1 Browsing the AST: Visitor pattern

The visitor pattern\textsuperscript{10} is used in object-oriented programming and software engineering to separate an algorithm from an object structure on which it operates. In this way, new operations can be added to existing object structures without modifying those structures. However, the visitor pattern is more limited than conventional virtual functions. It is basically not possible to create visitors for objects without adding a small callback method inside each class, because of the double dispatching (see below paragraph Details).

Details

“A user object receives a pointer to another object which implements an algorithm. The first is designated the “element class” and the latter the “visitor class.” The idea is to use a structure of element classes, each of which has an accept() method taking a visitor object for an argument. visitor is a protocol (interface in Java) having a visit() method for each element class.
The `accept()` method of an element class calls back the `visit()` method for its class. Separate concrete `visitor` classes can then be written to perform some particular operations, by implementing these operations in their respective `visit()` methods. (See the Figure 8 on page 11) [...] The visitor pattern also specifies how iteration occurs over the object structure. In the simplest version, where each algorithm needs to iterate in the same way, the `accept()` method of a container element, in addition to calling back the `visit()` method of the visitor, also passes the visitor object to the `accept()` method of all its constituent child elements." [10]

![Visitor design pattern diagram.](image)

Figure 8: Visitor design pattern diagram. [4]

Hopefully in Java, since Java 1.2, we can use the reflection mechanism [9] & [1]. Thanks to this, visited Objects no longer have to be modified and ignore being visited, the `accept()` method disappears. And any object becomes visitable, if a default method is added.

The visitor pattern with the reflection mechanism seems to be the best practice to manipulate the Abstract Syntax Tree (AST) what corresponds to the model inside the generator and produce the different output files of the generator according to the input TVL model.

7 Requirements

7.1 TVL extensions

This project requires some extensions of the Text-based Variability Language (TVL), like String or File attributes to refer specific source code folders that must be included or not. As well as the integration of a `visitor` component which must be able to walk through the structure of the normalized TVL model. Basically, the parser needs to provide an access to the root of the normalized TVL model.
References


A Description of the functionalities

The functionalities identified in the subsection 5.1 on page 5 are explained in more detail in this appendix. The description is based on a sequence diagram for each functionality and shows interactions between users and the system. For details about interaction inside the system see the Annex [B] on page [15].

**Generate configurator** A generation of configurator parts is initiated by a user.

1. The user starts the system and provides the system with a TVL model.
2. The system parses this TVL model.
3. The system generates a part of the configurator according to the parsed TVL model.
4. The system informs the user the task is successfully done.

See the Figure 9 on page 13.

![Sequence diagram of Generate configurator](image)

**Edit configuration** An edition of a configuration is initiated by a user.

1. The user starts the system and provides the system with the model he wants to configure.
2. The system starts the GUI system for the specified model.
3. The GUI system displays the specified model.
4. The user does a choice.
   - The user must do a choice until a configuration is complete.
5. The system forwards the user’s choice to the GUI system.
6. The GUI system validates the user’s choice and updates the display.
7. The system is informed the task is successfully done and forwards it to the user.
8. The user gets the configuration from the GUI system.
9. The system gets the configuration from the GUI system.
10. The GUI system sends the current configuration.
11. The system translates the received configuration and sends it to the user.
12. The user stops the edition process.
13. The system turns off the GUI system.
A DESCRIPTION OF THE FUNCTIONALITIES

14. The user submits the configuration to check it. See the functionality: Check configuration.

See the Figure 10 on page 14

**Check configuration**  A checking of a configuration is initiated by a user.

1. The user starts the system and provides the system with the configuration.
2. The system validates the configuration and informs the user the task is successfully done.

See the Figure 11 on page 14

**Create CAM standalone model**  A creation of a CAM standalone model is initiated by a user.
B DESCRIPTION OF THE FUNCTIONALITIES INSIDE THE SYSTEM

1. The user submits a configuration to check it. See the functionality: Check configuration.
2. The user starts the system and provides the system with a configuration.
3. The system lists the source directories required by the specified configuration in a filepath.
4. The system produces a makefile.
5. The system informs the user the task is successfully done.

See the Figure 12 on page 15.

Figure 12: Sequence diagram of Create CAM standalone model

B Description of the functionalities inside the system

This appendix describes interaction inside the system for the functionalities identified in the subsection 5.1 on page 5. The description is based on a sequence diagram for each functionality.

Generate configurator A generation of a configurator parts inside the system is managed by the Generator.

1. The generator executes the TVLparser to parse the input TVL model.
2. The generator gets the root element of the normalized TVL model from the TVLparser.
3. The generator produces the dimacs files by executing the appropriate method from the TVLparser.
4. The generator produces the rest of its outputs according to the same following pattern.
   The generator asks the creation of the appropriate visitor to produce one of its output from the Visitor.
   The generator executes this visitor on the root.

See the Figure 13 on page 16.
B DESCRIPTION OF THE FUNCTIONALITIES INSIDE THE SYSTEM

Check configuration  A checking of a configuration inside the system is managed by the Checker of the Configurator.

1. The Configuration delegates the checking to the Checker.
2. The Checker inits a SATsolver with the dimacs constraints.
3. The Checker translates the configuration into the dimacs format.
4. The Checker calls the SATsolver to validate the configuration translated in the dimacs format.
5. The Checker receives the result from the SATsolver and forwards it to the Configurator.

See the Figure 14 on page 17.

Create CAM standalone model  A creation of a CAM standalone model inside the system is managed by the Builder of the Configurator.
B DESCRIPTION OF THE FUNCTIONALITIES INSIDE THE SYSTEM

![Sequence diagram of Check configuration inside the system](image1)

Figure 14: Sequence diagram of Check configuration inside the system

1. The Configurator validates the configuration. (See the functionality: Check configuration)
2. The Builder writes the Filepath according to the component list.
3. The Builder produces the Makefile and calls the ScriptExecutor to run the custom Perl script with the appropriate arguments.
4. The ScriptExecutor executes the custom script with the specified arguments and informs the Builder that it’s successfully done.
5. The Builder forwards the result to the Configurator.

See the Figure 15 on page 17

![Sequence diagram of Create CAM standalone model inside the system](image2)

Figure 15: Sequence diagram of Create CAM standalone model inside the system