A4WSN – the Programming Framework and Implementation details

Ivano malavolta

i.malavolta@vu.nl

Abstract

This document describes the programming framework and the implementation details of A4WSN, the architecture-based modelling framework supporting the development and analysis of Wireless Sensor Networks (WSNs).

Keywords: WSN, Software Architecture, MDE

1. The A4WSN Programming Framework

Figure 1 shows an overview of the A4WSN programming framework. All the boxes within the programming framework represent the various components of the generic programming workbench, whereas the $C_1..C_n$ and $A_1..A_n$ boxes represent third-party code generation and analysis plugins, respectively. Third-party plugins extend the *Code Generation Manager* and *Analysis Manager* components which provide the needed extension points and they communicate with all the other components of the programming framework (for the sake of clarity we do not show those connectors in the figure). In the following we will discuss the facilities and duties of the various components of the generic A4WSN programming framework, an overview of their implementation details is provided in Section 2.

Models

The central element of the programming framework is the Models repository that stores all the WSN models developed by architects and designers. Indeed, stored models can conform to any A4WSN modelling language, which are SAML, NODEML, ENVML, MAPML, and DEPML. The models repository can be realised in different ways. For instance it may directly rely on the file system of the machine running the A4WSN platform (this is the solution implemented in the current version of the A4WSN tool), it may point to resources stored in the cloud

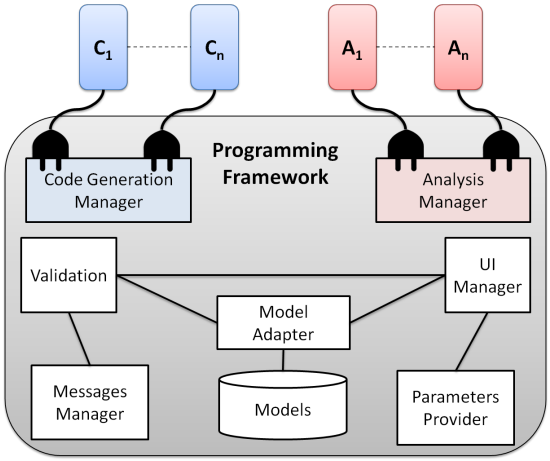

Figure 1: The A4WSN programming framework

or it may refer to some in-memory models representation. If on one side this feature of the models repository is very flexible in terms of resources consumption and localisation, on the other side it opens for possible problems of interoperability between all the other components of the A4WSN programming framework. This is exactly why the Model Adapter component exists.

Model Adapter

The model adapter is a component which abstracts the nature of the models repository to the other components of the A4WSN programming framework. The model adapter is composed of a set of connectors (each of them tailored to a specific models storage type) that expose a common interface to all the other components to access various elements of the models in a homogeneous way. Also, the Model Adapter component has a built-in model transformation, called ***Merger***, that can merge linked models defined in the A4WSN modelling environment. If we consider ***Merger*** as a function, it can be defined as follows:

$$Merger\colon MM_{SAML}\ x\ MM_{NODEML}\ x\ MM_{ENVML}\ x\ MM_{MAPML}\ x\ MM_{DEPML} \rightarrow MM_{merge}$$

where each MM_x is the metamodel of the x modelling language, where x can vary between $SAML$, $NODEML$, $ENVML$, $MAPML$, $DEPML$, and MM_{merge} is the union of all the MM_x metamodels. In other words, ***Merger*** takes as an input an instance of each modelling language defined in the A4WSN modelling environment and provides a single model conforming to a unique metamodel as

an output. The reason behind the existence of the Merger transformation is that currently many approaches and tools for code generation and analysis assume to have a single model as an input, rather than a set of models conforming to different languages. In order to alleviate this issue with current approaches and tools (which could have hampered the usefulness of the whole A4WSN platform), we decided to implement the Merger as an internal transformation to merge separate models into a single one. Merger can be executed at any time by plugin developers by calling a dedicated Java method.

Validation

The Validation component executes all the operations to validate A4WSN models:

- it checks whether one of the A4WSN models conforms to its corresponding metamodel;
- it executes all the OCL constraints defined in each metamodel within the A4WSN platform and checks whether they are satisfied or not;
- if defined, it executes the additional OCL constraints that are defined in some code generation or analysis plugin and checks whether they are satisfied or not.

The result of a validation operation is composed of four main elements: (i) a boolean value representing whether the involved model passes all the checks listed above, (ii) a set of informative messages that describe the result of the validation in a human-readable way, (iii) a set of in-memory representations of all the elements in the models which do not satisfy some of the checks listed above, and (iv) a set of actions that can be executed by the A4WSN platform as a quick fix of the identified violations (quick fix operations can be defined in the plugins extending the A4WSN platform).

The Validation component communicates with Model Adapter in order to access various elements of the models to be validated. Also, it communicates with the Messages Manager and the UI Manager components to show the informative messages belonging to M to the user and to highlight the elements in their graphical editor violating the constraints, respectively.

Messages Manager

The Messages Manager component serves to graphically show informative messages to the user. A4WSN supports three kind of informative messages which are *error, warning and information*. Plugin developers can decide the type of each message to be shown, depending on its severity. Each message is defined as a couple $< K, T >$, where K represents the type of message (i.e., error, warning, or information) and T represents the textual content of the message in a human readable way.

UI Manager

The UI Manager component is responsible for the main facilities interacting with the user interface of the A4WSN platform[1]. The UI Manager component provides all the graphical facilities to interact with the plugins and elements of the A4WSN platform, which are:

- *Code Generation Engines View*: a dedicated view showing a list of all the available code generation engines (with their description, icon, name, etc.), together with their management facilities, such as code generation activation, code generation results viewer, etc.;
- *Analysis Engines View*: a dedicated view showing a list of all the available analysis engines (with their description, icon, name, etc.), together with their management facilities, such as analysis activation, analysis results viewer (significantly different from the code generation results viewer), etc.;
- *Code Generation Contextual Menu*: a contextual menu that triggers the execution of a code generation engine. A contextual menu is associated to each model of the A4WSN modelling environment;
- *Analysis Contextual Menu*: a contextual menu that triggers the execution of an analysis engine. A contextual menu is associated to each model of the A4WSN modelling environment;
- *Validation Trigger*: a contextual menu and a dedicated button in the graphical editor of each model of the A4WSN modelling environment that triggers the validation of the current model. Optionally, the user can identify which

[1] Also the Messages Manager interacts with the UI of the A4WSN platform, however its impact to the UI is much more limited than that of UI Manager.

plugin contains additional constraints to be checked. The results of the triggered validation are managed by the Messages Manager component;

- *Code Generation and Analysis Progress Feedback*: provides an element in the UI that graphically shows the progress of the triggered code generation or analysis. A4WSN provides two types of progress feedback, a progress bar for activities in which all the steps are known a priori and a round indicator for activities with an unknown length.

- *Plugin Additional Parameters View*: provides a dedicated view in which users can provide additional parameter to be passed to the code generation or analysis engine being triggered. Plugin developers can specify the number, name, and type of those parameters by using a specific extension point.

Parameter Provider

Parameter Provider component manages the additional parameters that a code generation or analysis plugin may require for carrying on its activities. As previously mentioned, additional parameters are defined by using a specific extension point of the A4WSN programming framework; each parameter is defined as a triplet $< name, T, default >$, where $name$ is the unique name of the parameter, T is the type of the parameter, and $default$ is the optional default value of the parameter. Available parameter types are listed below.

- *String*: a textual value;

- *Integer*: an integer numerical value;

- *Float*: a decimal numerical value;

- *Boolean*: a boolean value;

- *Local Resource*: a file in the local file system of the user, it is referenced by its path in the file system;

- *Remote Resource*: a resource in the cloud that can be accessed by a standard HTTP GET request and is referenced by its URL.

Once the user has provided the values of the additional parameter of a code generation or analysis engine, the Parameter Provider component makes them available to the plugin realizing the engine so that it can access them before actually executing the activity which is being triggered by the user.

Code Generation Manager

The Code Generation Manager provides a set of facilities for managing code generation engines and the extension point that is used by code generation plugin developers (see Section 1 for more details). For instance it checks which plug-ins are currently extending its extension point and makes their facilities available to the end user. It includes all the registered code generation plugins into the *Code Generation Engines View* of the UI Manager. It loads plugins into the contextual menus of the A4WSN modelling environment. It automatically triggers the validation operations defined by the plugins before executing the actual code generation operation. Also, the Code Generation Manager component exposes a common *Java API* to plugin developers, so that they can easily interact with all the other components of the A4WSN programming framework. For example, it allows developers to access elements of the models in the Models Repository to push messages to the end user via the Messages Manager and it makes the additional parameters provided by the end users accessible directly as Java objects.

Analysis Manager

The internal logic of the Analysis Manager component is analogous to that of Code Generation Manager. The only difference is that it is designed for analysis plugins, rather than for code generation plugins. Due to its similarity to Code Generation Manager, the reader can easily grasp its functioning from the description of the latter, so we will not describe the Analysis Manager component in this paper. In Section 1 we discuss the extension points that are available to code generation and analysis plugin developers.

Extension Points

The concept of extension point is nicely described in the Eclipse Wiki[2], it says that *the extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plug-ins that want to connect to that extension point must implement that contract in their extension. The key attribute is that the plug-in being extended knows nothing about the plug-in that is connecting to it beyond the scope of that extension point contract. This allows plug-ins built by different individuals or companies to interact seamlessly, even without their knowing much about one another.* The last part of the Eclipse

[2] `http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points\%3F`

definition of extension point says exactly what we are demanding to the WSN research community, i.e., not to rebuild the wheel by focussing on modelling languages, graphical editors, etc., but rather to focus on code generation and analysis of WSN applications by developing A4WSN plug-ins.

Table 1 shows various extension elements that can be set by third-party developers with their plugins. For each element we specify its name, whether it belongs to the code generation (column titled *CG*) or analysis extension point (column titled *A*), and a description about how it will be used by the generic A4WSN programming framework.

The extension points defined in the A4WSN programming framework are used to group code generation and analysis engines into two different groups, so that the end user knows where those engines can be found. Also, they are used to provide a common, standard behaviour to various engines that may be defined upon the A4WSN modelling environment. Both the Code Generation Manager and Analysis Manager provide a standard management of the workflow that must be followed when executing those engines. For example, they automatically call the pre-actions defined by using the *Pre Action* element of the previously defined extension point (the same holds for the post-action). Automatically manage the success and error messages to be shown after the execution of either a code generation or analysis operation, automatically update the UI of the modelling framework depending on the available plugins extending A4WSN, etc. Moreover, since plugin developers must comply with to the extension points defined in the A4WSN programming framework, they will be more keen to provide engines that are straightforward to integrate and with common basic functionalities, thus easier to use by end users.

2. Implementation of A4WSN

We make the current prototype of the proposed approach available to the community as an open-source product with MIT license in order to allow other researchers to use the A4WSN modelling languages as well as the programming framework described in the previous section. The current prototype of A4WSN can be downloaded from the A4WSN website (http://a4wsn.di.univaq.it).

We implemented the proposed approach by extending the **Eclipse** platform[3].

[3]Eclipse project Web site: www.eclipse.org.

Element	CG	A	Description
Name	✓	✓	The name of the engine being provided which will be shown in the engines view and contextual menus.
Icon	✓	✓	An image icon of the engine being provided which will be shown in the engines view and contextual menus.
Description	✓	✓	A textual description engine being provided which will be shown in the engines view.
Network Access	✓	✓	A boolean for declaring whether the engine uses the network for its operations which will be shown in the engines view.
Operation Time	✓	✓	An estimation of the time needed to complete the operation being defined which will be shown in the engines view.
Target Languages	✓	-	A list of the target implementation languages which will be shown in the engines view and contextual menus.
Target Path	✓	-	The path in the file system (local to the location of the plugin) to which the generated code will be saved.
Analysis Type	-	✓	A list of the properties that will be checked during the analysis operation (e.g., performance, security, etc.); it will be shown in the engines view.
Keep Intermediate	-	✓	A boolean value (optionally a path in the file system) to specify whether (and where) the analysis engine keeps possible intermediate resources.
Additional Parameters	✓	✓	A list of parameter types definition that will be used by the Parameter Provider component of A4WSN.
Validation Constraints	✓	✓	A list of OCL constraints, together with their informative messages and quick fix operations that must be used by the Validation component of A4WSN.
Pre Action	✓	✓	A reference to a Java class defining the method that will always be called before executing the engine being provided.
Post Action	✓	✓	A reference to a Java class defining the method that will always be called after the engine being provided is executed.

Table 1: Elements of the extension points for code generation or analysis plugins

Eclipse is an open-source development platform comprised of extensible frameworks and tools for building, deploying and managing software across the life cycle. We decided to use Eclipse as starting point for our modelling environment for three main reasons. First, many extensions already exist covering some aspects of our approach (e.g., graphical syntax definition for newly created languages, models persistence support, etc.). Second, its plugin architecture allows us to provide a set of extension points that other developers can use to extend our modelling framework,. Third, the Eclipse community is widely spread throughout the world, raising the possibility of adoption of our modelling environment.

For what concerns the modelling languages, model-driven engineering techniques are used to define their concepts, and their modelling environment. More specifically, we specified the static semantics of the languages by means of their underlying metamodels. Those metamodels are defined by using the **Eclipse modelling Framework** (EMF)[4], that is a Java framework and code generation facility for building tools and other applications based on a metamodel. The concrete syntax of the modelling languages has been defined by using the **Graphical modelling Framework** (GMF)[5], a model-driven approach to generate graphical editors in Eclipse.

The intermediate modelling languages (i.e., MAPML and DEPML) are technically called weaving models. Weaving models are special kinds of models for defining relations among other models and to establish semantic links among model elements. Weaving models have been successfully used in many fields, such as software architecture [1] and software product lines [2]. We use the **Atlas Model Weaver (AMW)** [3] for managing those weaving models.

For what concerns the programming framework, we implemented it as a set of **Eclipse plugins**, each one implementing a single component of the programming framework, as it is depicted in Figure 1. Those plugins are implemented in Java and their dependencies are realized by means of the plugins management system provided by Eclipse. Each plugin declares the others it depends on and configuration parameters via a specific XML configuration file. The communication among plugins is handled by standard Java calls. Also, the code generation framework and the analysis framework provide two extension points dedicated to code generation and analysis plugins, respectively. The signatures of those extension points are defined in the same XML configuration files used for defining the

[4]EMF project Web site: http://www.eclipse.org/modeling/emf/.
[5]GMF project Web site: http://www.eclipse.org/modeling/gmf/.

dependencies between plugins, whereas their implementation is defined as Java classes referenced by the XML configuration files. For the sake of brevity, we do not provide the details on how the programming framework works and on how its plugins interact. A detailed description of those aspects can be found in the Eclipse plugin developer guide[6].

References

[1] I. Malavolta, H. Muccini, P. Pelliccione, D. Tamburri, Providing architectural languages and tools interoperability through model transformation technologies, Software Engineering, IEEE Transactions on 36 (1) (2010) 119 –140.

[2] K. Czarnecki, M. Antkiewicz, Mapping features to models: A template approach based on superimposed variants, in: GPCE, 2005, pp. 422–437.

[3] Didonet Del Fabro M., Bézivin J., Jouault F. and Breton E. and Gueltas G., AMW: a generic model weaver, in: Proc. of 1 re Journ e sur l'Ing nierie Dirig e par les Mod les, Paris, France. pp 105-114, 2005.

[6] Eclipse Platform Plug-in Developer Guide: http://help.eclipse.org/helios/index.jsp