15

An Experiment Description Language for Supporting Mobile IoT Applications

Kostas Kolomvatsos, Michael Tsiroukis and Stathes Hadjiefthymiades

Pervasive Computing Research Group,
Department of Informatics and Telecommunications,
National and Kapodistrian University of Athens,
Panepistimiopolis, Ilissia, 15784, Greece

Abstract

Mobile IoT applications consist of an innovative field where numerous devices collect, process and exchange huge amounts of data with central systems or their peers. Intelligent applications could be built on top of such information realizing the basis of future Internet. For engineering novel applications, experimentation plays a significant role, especially, when it is performed remotely. Remote experimentation should offer a framework where experimenters can efficiently define their experiments. In this chapter, we focus on the experiments definition management proposed by ***Road-, Air- and Water-based Future Internet Experimentation*** (RAWFIE). RAWFIE offers, among others, an experimentation language and an editor where experimenters can remotely insert their experiments to define actions performed by the nodes in a testbed. RAWFIE proposes the ***Experiment Description Language*** (EDL) that provides the elements for the management of devices and the collected data. Commands related to any aspect of a node behavior (e.g., configuration, location, task description) are available to experimenters. We report on the EDL description and the offered editors and discuss their key parts and functionalities.

15.1 Introduction

The ***Internet of Things*** (IoT) assumes the pervasive presence of numerous devices in the environment that are capable of performing simple processing tasks by involving multiple interactions among them. Such devices are wirelessly connected and adopt unique addressing schemes to be uniquely identified in the network. Objects make themselves recognizable and they obtain intelligence by taking or enabling context related decisions thanks to the fact that they can communicate information about themselves and they can access information that has been aggregated by other things, or they can be components of complex services [38]. The number of Internet-connected devices surpassed the number of human beings on the planet in 2011, and by 2020, Internet-connected devices are expected to a number between 26 and 50 billion. For every Internet-connected PC or handset there will be 5–10 other types of devices sold with native Internet connectivity [28]. Novel applications can be built on top of the vast network to improve the services offered to end users and, thus, to improve their quality of living. Mobile IoT involves devices capable of moving in the environment and record the ambient information. A typical example is the adoption of ***Unmanned Vehicles*** (UV). UVs can be categorized into: ***Unmanned Ground Vehicles*** (UGVs), ***Unmanned Aerial Vehicles*** (UAVs) and ***Unmanned Surface Vehicles*** (USVs). UVs act autonomously and can carry a set of sensors and communicate each other as well as with a central system where they can transfer the data that they record during their movement.

In this context, the research and technical challenges towards the development of a smart World are many. These challenges call for efficient solutions either *horizontally* (application neutral) or *vertically* (application dependent). Mobile IoT should overcome the vertical oriented legacy architectures and lead to open systems and integrated environments that will support intelligent applications on top of contextual knowledge collected/produced by autonomous nodes. The aim is to create innovative ecosystems of moving devices like UVs. An efficient means for building high quality applications is *remote experimentation*. Remote experimentation has already adopted in domains like education [1, 7, 13, 21]. It involves a real experiment with real equipment that is controlled remotely through the Internet. Remote experimentation can offer many advantages as physical experimentation is expensive, difficult to maintain and restricted to specific areas. Usually, *testbeds* are adopted to host a set of devices that will execute an experiment. A ***testbed*** is a platform/environment where hardware (e.g., a number of devices) is available to perform transparent and replicable testing of tools or technologies. Testbeds can be located around the World and host devices belonging to multiple types, if possible.

The ***Road-, Air- and Water-based Future Internet Experimentation*** (RAWFIE) platform comes to offer remote experimentation functionalities to the interested researchers and professionals. RAWFIE delivers a framework for interconnecting numerous testbeds over which remote experimentation will be realized. The RAWFIE platform originates in a European Union-funded (H2020 call: FIRE+ initiative) project which focuses on the mobile IoT paradigm and provides research and experimentation facilities through the ever growing domain of unmanned networked devices (vehicles). The purpose of the proposed structure is to create a federation of different network testbeds that work together to make their resources available under a common framework. Remote experimentation is realized on top of real devices. These devices have specific characteristics that may vary according to their type (e.g., UGVs, UAVs, USVs). Devices characteristics may vary even devices belong to the same category as they come from their manufacturers.

In remote experimentation, there is a gap between experimenters and devices realizing an experiment. Experimenters are researchers or professionals and may not be aware of the characteristics of the devices. Experimenters, likely, are not aware of the low level instructions that should be transferred to the devices during the execution of the experiments. To cover this gap and serve non experienced experimenters/professionals, RAWFIE offers an abstraction of the underlying functionalities. This abstraction is realized by a ***Domain Specific Language*** (DSL). A DSL is a language designed for a specific field of applications. Its aim is to solve problems related to a highly focused field of research. DSLs target to more specific tasks than classic programming languages. They provide expressions for describing parameters of a domain and they have a concrete syntax. A number of semantics are adopted to lead to the automatic generation of specific tools important for the creation of the final code [17]. The most significant advantage of the DSLs usage is that they provide the opportunity to non-experienced users to write more easily domain specific programs [20]. These programs are not dependent on the underlying platform, thus, providing an additional advantage. RAWFIE offers an ***Experiment Description Language*** (EDL), i.e., a DSL, and two editors (textual and visual) devoted to assist non-experienced users to easily define their experiments. A code generation component is responsible to translate each experiment expressed in the EDL into the information transferred to mobile nodes. Hence, RAWFIE efficiently interconnects experimenters coming from various domains with the nodes present in numerous testbeds.

The rest of the chapter is organized as follows. Section 15.2 is devoted to the description of the problem while Section 15.3 presents the related work. Section 15.4 discusses our approach for creating a DSL for abstracting the complexity of the UxVs characteristics and Section 15.5 reports on the technical details. Section 15.6 presents a case study where we create and launch an experiment with the proposed tools and Section 15.7 discusses our future research directions. Finally, in Section 15.8, we conclude our chapter.

15.2 Problem Statement

The definition of an experiment on top of a number of devices located in testbeds around the Globe involves the creation of a script containing commands that will be executed by the devices. Devices should receive the instructions defined in the script and move in the environment towards the execution of the experiment. For instance, an experiment may instruct a group of UVs to move around an area and collect data related to environmental conditions (e.g., temperature, humidity). In this scenario, experimenters should know the low level characteristics of the devices (e.g., commands for defining navigational instructions to UVs). However, this is not the usual case. It is difficult for experimenters to know the low level commands especially when they are working in a completely different domain. Imagine a researcher working in biomonitoring and the effects of environmental conditions in humans' health. The researcher does not have any technical knowledge on the functionalities provided by the UVs, however, he/she wants to instruct the devices to perform some processing tasks. The problem is more intense when we take into consideration that a testbed may incorporate many different devices with different characteristics. Experimenters cannot be aware of the different sets of commands to handle the heterogeneity of the devices. Due to the wide range of devices and technologies that testbeds could incorporate, a number of different commands could be adopted to instruct devices to execute experiments.

The above discussion shows the need for an abstraction in the underlying technologies. Such an abstraction will give the opportunity to experimenters to use the devices transparently and define commands for UVs in a more user friendly way. Hence, even non-experienced experimenters can use the provided platform and define their experiments that will be executed to any available testbed. In general, users, not having a lot of experience with programming languages, are not able to develop efficient software components like experiments for mobile IoT. In this case, *Model Driven Engineering* (MDE) can provide a lot of advantages not only to under-experienced programmers but also to proficient ones that are unfamiliar with the specific domain. MDE is a software development methodology for creating models for a specific domain. MDE technologies offer a promising approach to address the inability of the third generation languages to express domain concepts effectively [32]. The aim of MDE is to increase efficiency in developing applications. DSLs follow the principles of the MDE development and can provide a number of advantages in cases where domain programming knowledge is limited [22, 35]. DSLs target to more specific tasks than classic programming languages. They provide expressions for describing parameters of a domain of interest and they have a concrete syntax. A number of semantics are used in order to lead to the automatic generation of specific tools important for the creation of the final code [17].

RAWFIE offers the EDL, that provides a terminology for defining experiments for mobile IoT. The EDL offers an abstraction for any aspect of an experiment like the necessary metadata, statements, commands related to the devices, group of devices management and so on. The EDL terminology is invoked through the provided *Experiment Authoring Tool* (EAT). Two editors are provided: the ***textual*** and the ***visual***. Editors are built on top of the EDL and incorporate all the necessary functionalities like those originated in typical IDEs as well as functionalities related to the compilation and validation of the defined experiments.

15.3 Background and State of the Art

A number of research efforts deal with the devise and development of *Vehicular Ad Hoc Network* (VANET) testbeds for performing diverse applications (e.g., accident warning systems, traffic information control and prevention systems, pollution and weather monitoring, etc.). C-Vet [8] stands for the vehicular testbed developed in the University of California at Los Angeles (UCLA) campus offering both Vehicle to Vehicle (V2V) and Vehicle to Infrastructure (V2I) connectivity. The testbed is composed of 60 vehicles that circulate in the UCLA campus in order to support extended applications and services. CarTel [16] is a testbed developed by Massachusetts Institute of Technology (MIT) that has been active in Boston and Seattle. CarTel is comprised of six vehicles equipped with sensors and communications units that feature Wi-Fi (IEEE 802.11b/g) and Bluetooth. This testbed provides an important insight on how to handle intermittent connectivity, and how feasible this kind of connectivity is to explore a class of non-interactive applications. SAFESPOT [31] is a testbed that was run for 4 years in six cities across Europe. It uses vehicles equipped with OBUs, RSUs and Traffic Centres (communicating through Wi-Fi) to centralize traffic information and forward safety-critical messages. The project's goals were to: a) use the infrastructure and the vehicles as sources and destinations of safety-related information and develop an open, flexible and modular architecture and communication platform, b) Develop the key enabling technologies: ad-hoc dynamic network, accurate relative localisation, dynamic local traffic maps, c) Develop and test scenario-based applications to evaluate the impacts on road safety, d) Develop and test scenario-based applications to evaluate the impacts on road safety and e) Define a sustainable deployment strategy for cooperative systems for road safety, evaluating also related liability, regulations and standardisation aspects. HarborNet [2] is a real-world testbed for research and development in vehicular networking that has been deployed successfully in the sea port of Leixoes in Portugal. The testbed allows for cloud-based code deployment, remote network control and distributed data collection from moving container trucks, cranes, tow boats, patrol vessels and roadside units, thereby enabling a wide range of experiments and performance analyses.

DSLs have attracted a lot of attention in various application domains as they provide abstraction in the definition of applications oriented to a specific research field [14]. Every DSL has special characteristics and their size varies according the domain of application. Normally, DSLs are small in length and cover only the essential features and concepts of the domain under consideration [25], however, they are characterized by expressiveness [22]. This approach keeps the length of the notation small, thus, increasing the abstraction level. DSLs are more declarative or descriptive than legacy programming languages [36]. The design of a DSL involves the study of the domain under consideration and the identification of the most important concepts of that domain. The semantics of the domain should be implicit in the language notation [15].

For a theoretical survey in DSLs, the interested reader should refer in [25] while an empirical study on the use of a DSL in industry is presented in [12]. A number of contributions discuss the advantages of DSLs [18, 33, 34, 36] while a survey on the process for developing a DSL is described in [22]. In general, DSLs lead to easy maintenance of potential modifications, increase flexibility and productivity. DSLs are adopted to a set of research domains. In robotics, DSLs focus on increasing the level of automation, e.g., through code generation, to bridge the gap between the modeling of robotics and implementation. In [24], the authors survey the corresponding literature and classify a number of publications in the robotics field. DSLS are also adopted in banking [3], telecommunications [6], web services definition [12], autonomic computing [19]. DSLs are already adopted in a number of research projects like IPAC[1] and PoLoS[2]. The IPAC (Integrated Platform for Autonomic Computing) aims at delivering a middleware and service creation environment for developing embedded, intelligent, collaborative, context-aware services in mobile nodes. A DSL is implemented to support engineers to efficiently define applications that will be uploaded in mobile nodes. The PoLoS project aims to design specify and implement an integrated platform, which will cater for the full range of issues concerning the provision of *Location Based Services* (LBS). PoLoS proposes a DSL for 'annotating' LBSs that will be combined in the final workflow.

In [40], the authors demonstrate a framework to automate the generation of DSL testing tools. The presented framework utilizes Eclipse plug-ins for defining DSLs. Moreover, a set of tools concerning a translator, and an interface generator are responsible to map the DSL debugging perspective to the underlying *General Purpose Language* (GPL) debugging services. The aim is to present the feasibility and the applicability of debugging and testing information derived by a DSL in a friendly programming environment. A program transformation engine supporting the debugging process written in a DSL is described in [29, 39, 40]. The discussed approach concerns the methodology of generating a set of tools necessary to use a DSL from a language defined in a specific grammar. Such tools are: the editor, the compiler and the debugger [11]. This research effort focuses on issues related to the debugging support for a DSL development environment. The debugger is automatically generated by a language specification. Authors describe two approaches for weaving the debugger in conjunction with the *DSL Debugging Framework* (DDF) plug-in. The first approach is applicable when the aspect weaver is available for the generated GPL while the second approach involves the *Design Maintenance System* (DMS) [4] transformation and is applied when the aspect weaver is not available.

In [30], the authors describe a prototyping methodology for of *Domain Specific Modeling Languages* (DSMLs) on an independent level of the MDE architecture. They argue that the prototyping method should describe

[1]http://ipac.di.uoa.gr/
[2]http://polos.di.uoa.gr/

the semantics of the DSML in an operational fashion. For this, they use standard modeling techniques i.e., *Meta Object Facility* (MOF) [23] and *Query/View/Transformations* (QVT) Relations [27]. By combining this approach with existing metamodel-based editor creation technologies they enable the rapid and cost free prototyping of visual interpreters and debuggers. Authors utilize the *Eclipse Modelling Framework* (EMF) which is similar to MOF and using the Ecore metamodel of a DSML they can generate the DSML plug-in with EMF. The created plug-in provides the basis for the creation, access, modification, and storage of models that are instances of the DSML.

A logic programming based framework for specification, efficient implementation, and automatic verification of DSLs, is presented in [10]. Their proposal is based on Horn logic and, eventually, constraints to specify semantics of DSLs. The semantic specification serves as an interpreter or more efficient implementations of the DSL, such as a compiler, can be automatically derived by partial evaluation. The executable specification can be used for automatic or semi-automatic verification of programs written in a DSL as well as for automatically obtaining conventional debuggers and profilers. The syntax and semantics of the DSL are expressed through Horn logic. The Horn logic syntax and semantics are executable leading to the automatic definition of an interpreter. The authors in [10] present their approach and give some examples indicating the efficiency of the discussed methodology.

15.4 The Proposed Approach

15.4.1 The RAWFIE Platform

The purpose of the RAWFIE initiative is to create a federation of different testbeds that will be combined to make their resources available under a common framework. Specifically, RAWFIE aims at delivering a unique, mixed experimentation environment across the space and technology dimensions. RAWFIE will integrate numerous testbeds for experimenting in vehicular (road), aerial and maritime environments. The basic idea behind the RAWFIE effort is the automated, remote operation of a large number of robotic devices (UGVs, UAVs, USVs) for the purpose of assessing the performance of different technologies in networking, sensing and mobile/autonomic application domains. RAWFIE features a significant number of UVs for exposing to the experimenter a vast test infrastructure. All these items are managed by a central controlling entity which is programmed per case and fully overview/drive the operation of the respective mechanisms (e.g., auto-pilots, remote controlled ground vehicles). Internet connectivity will be extended to the mobile units to enable the remote programming (over-the-air), control and data collection. Support software for experiment management, data collection and post-analysis is virtualized to enable experimentation from everywhere in the world. The vision of *Experimentation-as-a-Service* (EaaS) is promoted through RAWFIE. The IoT paradigm is fully adopted and further refined for support of highly dynamic node architectures.

The RAWFIE architecture consists of tree tier design patterns. Each tier is separated in different software elements, each one providing a different functionality. The components are implemented with standard interfaces for safe interconnection between them. The discussed tiers are: i) the *front-end tier*, ii) the *middle tier* and iii) the *data tier*. The front end tier includes the services and tools that RAWFIE provides to experimenters to define and perform the experimentation scenarios. The RAWFIE *Web portal* provides to users, a web interface to federation resources and services. The user friendly environment of the portal makes experimenters creating straightforward successful experiment scripts. The front end tier has an authorization component, for checking the authorization of a user by his/her credentials. The *Testbed and Resource Discovery* component shows the availability of the testbed and the resources respectively while running. The *Experimentation Suite* is consisting of five tools and are the following: i) the *Monitoring tool* – it manages the presentation of the information needed for monitoring the status of the nodes and the data collected during the experiments; ii) the *Launching tool* – it is informed for the end of an experiment's execution to initiate the next booked scenario in the case of the entire use of a testbed or it is invoked (manually) to start an script that experimenters desire; iii) the *Booking tool* – it allows experimenters to book a spatiotemporal interval for running their experiments, thus, providing automatic coordination in the use of the testbed resources among experimenters; iv) the *Visualization tool* – it interconnects with the *Visualization Engine* of the middle tier receives the resource traces. The resource traces are graphically displayed to the web interface; v) the *Authoring tool* – it includes all the necessary mechanisms to allow access of the experimenters in the RAWFIE experimentation suite and the available EDL editors.

The RAWFIE middle tier is the layer that lies between the UVs testbeds and the experimenters (front-end tier). It provides the software interfaces needed, and includes useful software components related to security, trust, control and visualization aspects. This tier provides the infrastructure which facilitates the creation and integration of applications in the RAWFIE platform. It provides uniform, standard, high-level interfaces to the application developers and integrators so that the applications can be easily composed and reused. It will supply a set of common services to perform various general purpose functions in order to hide the distributed nature of the testbeds and facilitate the collaboration between different applications. The middle tier is consisted of the following modules: i) the *Experiment Validator* – it validates the experiment scenario to avoid syntactic and semantic errors. For instance, if the experimenter requests more resources than the available ones in the selected timeslot in the specified testbed site, the validator will avoid the execution of the experiment and send error message to the experimenter; ii) the *Experiment Controller* – it provides functionalities for the automatic control of the executed experiments according to the defined scripts; iii) the *Visualization Engine* – it is responsible for gathering sensing information from the UVs, processing the data and finally forwarding them to the visualization tool of the front-end tier; iv) the *Testbed directory* – it includes information relevant to the testbeds and resources (i.e., location, facilities) as well information on the capabilities of a particular resource and its requirements for executing experiments e.g., in terms of interconnectivity or dependencies; v) the *Data Collection and Analysis module* – it is responsible for the data collection and data the analysis-processing. Furthermore, it stores the measurement streams in the Data Storage components of the RAWFIE infrastructure. RAWFIE also provide a large, secure, cloud-based central repository in which collected data can be anonymized and made available to users; vi) the *Launching Service* – it provides functionalities related with the automatic and the manual launch of an experiment; vii) the *Booking Service* – it is adopted for performing bookings in the available testbeds and resources; viii) the *System Monitoring Service* – it secures that the platform works properly and identifies any potential error in the RAWFIE framework.

Finally, the data tier is in charge of ensuring data persistence. All the data elements and the code repos are stored to *Data Storage and Code Repositories* and servers to the Cloud, respectively.

15.4.2 The RAWFIE EDL

The Experiment Description Language (EDL) is a DSL for creating simple or more complex experimental scenarios for the IoT domain. The EDL is designed for the RAWFIE purposes aiming to help domain experts or non-experienced users (e.g., experimenters) to effectively create and handle

IoT remote experimentation. The major goal of the EDL is the provision of a high level of abstraction that shields experimenters from the complexities of the underlying implementation of the RAWFIE platform and the available devices. In the most interesting case, the EDL provides elements for handling resource requirements/configuration, location/topology information, task description, testbed-specific commands etc. Its syntax is simple and combines some common characteristics of well-known XML or legacy programming languages. The EDL is built with the help of the Xtext framework[3]. The following listing presents a small part of the proposed EDL grammar.

```
Experiment:
        'Experiment'
                metadata=MetadataSection
                (requirements=RequirementsSection)?
                (declarations=DeclarationsSection)?
                execution=ExecutionSection
        '~ Experiment'
;
/********** Metadata Section **********/
MetadataSection:
        'Metadata'
                met+=Metadata
        '~Metadata';
Metadata:
        'Name' name = ID
        (experimentVersion=Version)?
        (experimentDescritpion=Description)?
        (experimentDate=Date)?;
Version:
         'Version' ver=VER;
Description:
        'Description' name=ID;
Date:
        'Date' dat=DAT;
```

[3]https://eclipse.org/Xtext/

An experiment as realized through the EDL terminology is seen to have the following parts:

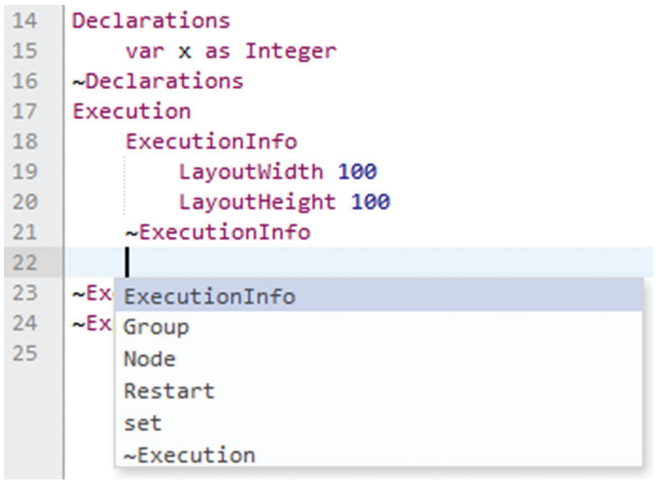
- ***Metadata section***. It contains generic information related to each experiment like the name, the date, etc. This information is important to define the necessary description for each experiment and, thus, to facilitate the efficient management of the available experiments.
- ***Requirements section***. It contains information related to the requirements of each experiment in terms of the testbed data, the location, the duration or the distance that the nodes should cover during the experiment execution. In addition, in this section, the experimenter should define the number of nodes that will be involved in the experiment and, thus, the RAWFIE platform is capable of knowing the needs for the experiments under consideration.
- ***Declarations section***. It concerns the necessary declarations like constants and variables declaration adopted to store data during the experiment execution. The discussed declarations are the key element to connect the experiment business logic with the data retrieved by UxVs and perform processing in a higher level than the device itself.
- ***Execution section***. It involves commands related to the management of the core business logic of each experiment. The EDL offers statements for the nodes or group of nodes management. Every aspect of nodes/groups behavior can be realized with specific terminology in the execution section. In addition, a number of statements are devoted to: (i) waypoints management; (ii) time line management (e.g., sequential or parallel execution); (iii) coordination management; (iv) control management (e.g., activation/deactivation of sensors); (v) configuration management (e.g., data management in each node); (vi) communication management (e.g., change in network interfaces).

It should be noted that 'typical' commands originated in legacy programming languages are also included in the EDL. Hence, assignments, conditionals statements (i.e., if, switch) and iterations (i.e., for, while) are also in place. In the following listing, we present a small part of an EDL script related to the definition of the behavior of a node. The 'Route' command instructs the node to follow a set of waypoints defined by multiple WP commands. Each waypoint is identified by three numbers: *time*, x, y and z coordinates. For instance, the command WP<3, 50, 22, 15> instructs the node, at time 3, to be at the location (50, 22), at height/depth 15.

```
Node
        ID node1
        Route[
                WP<1, 10, 12, 12>
                WP<3, 50, 22, 15>
                WP<15, 84, 42, 15>
                WP<18, 36, 22, 15>
                ]
        DataManagement
                Time 14 Algorithm average(history = 10)
        ∼DataManagement
        NodeCommunication
                NIC WiFi
        ∼NodeCommunication
        DataManagement
                Time 25 Algorithm average(history = 5)
        ∼DataManagement
∼Node
```

15.4.3 The EDL Textual Editor

On top of the EDL terminology, RAWFIE provides two editors: the textual and the visual editors. Both editors are provided as a Web application in a common interface separated in two parts. Editors are responsible to provide the necessary functionalities to the experimenters towards the creation, update, compilation and validation of their experiments. Editors are a collection of tools for defining experiments and authoring EDL scripts through the RAWFIE Web portal. Rich editing facilities are supported in the textual editor together with an advanced content assist and checking mechanism at syntax time. The EDL keywords are highlighted with different color while the code folding (only in the standalone version of the textual editor) functionality enables blocks of code to be hidden or expanded at will. Some of the provided functionalities of the textual editor are: (i) syntax coloring; (ii) content assist; (iii) validation and quick fixes; (iv) code completion; (v) error checking. A set of additional tools for syntactic and semantic validation are also available. The textual editor gives 'access' to the EDL concepts through which an experiment will be defined. Editors are synchronized and experimenters have the opportunity to define nodes routes and other related information directly on the map of the area under consideration (in the visual editor)



```
Declarations
    var x as Integer
~Declarations
Execution
    ExecutionInfo
        LayoutWidth 100
        LayoutHeight 100
    ~ExecutionInfo
    |
~Ex
~Ex
```

Figure 15.1 The content assist functionality of the EDL textual editor.

and the list of waypoints is immediately transferred to the textual editor. In Figure 15.1, we see a snapshot of the provided textual editor where the content assist functionality gives us hints about the upcoming commands that should be inserted in an experiment.

15.4.4 The EDL Visual Editor

The visual editor is an innovative and powerful tool for creating experiments in the RAWFIE authoring tool. The main goal of the visual editor is to provide a user friendly environment that simplifies the creation of an experiment by adopting 'typical' GUI functionalities (e.g., mouse actions). Experimenters have the opportunity to define basic UVs actions (e.g., waypoint definition) directly on the map. A set of tools, in the form of buttons, are available to the experimenters. Each button has a specific orientation i.e., nodes management (e.g., addition, deletion), nodes behavior definition (e.g., activation of sensors, define data management algorithms) while with the use of the mouse, experimenters can define the route of each UV in the area. Every node is depicted in the map with a different color to avoid confusion in the cases where an experiment involves multiple nodes. In addition, the visual editor gives the opportunity to experimenters to define the time when an action/movement should take place maintaining the spatio-temporal aspect of the experiment. It should be noted that both editors are synchronized while the error messages and warnings are presented in the textual editor area.

15.4.5 The Validator and the Generator

The EDL validator is responsible for performing syntactic and semantic analysis on the provided EDL scripts. The validation is performed on top of the proposed EDL model that is based on the EDL grammar. The validator accesses the provided script and identifies any semantic errors that could jeopardize the execution of an experiment. Specific constraints should be fulfilled when the experiment workflow is defined. These constraints are continuously checked by the proposed editors and in case some of them are validated to be false, errors will be presented to the experimenters through various means (e.g., with red color). The main responsibilities of the validator are: (i) it provides syntactic and semantic validation of each experiment workflow; (ii) it applies a set of constraints that should be met in order to have a valid experiment; (iii) it is capable of applying semantic checking for nodes communication, spatio-temporal management, sensing and data management.

RAWFIE also offers a code generation component. When no errors are present, the component has the opportunity to generate specific files e.g., part of the final code to be uploaded in the UVs. The code generation component takes as input the experiment workflow in terms of EDL commands and transforms them in the appropriate target language. This component conveys design and implementation issues that need to be handled in such a way that will help experimenters to avoid errors and development problems. The module receives commands from the available editors, data from the underlying model (the terminology of the EDL as depicted by the Ecore model) to create the experiment code/files.

15.5 Technical Details

15.5.1 The EDL Grammar

The EDL and the provided editors are built by adopting the Xtext framework[4]. The Xtext is a framework for the development of DSLs. It offers functionalities that let engineers to define their language using a powerful grammar. The grammar is the most important part of the Xtext framework and, actually, it is DSL by itself. The grammar aims to provide functionalities for describing the concrete syntax of a DSL (e.g., the EDL) and how it is mapped to an in-memory representation. The in-memory representation of the EDL is the semantic model. The semantic model is produced during the experiment definition by

[4]https://eclipse.org/Xtext/

the parser. The definition of the EDL with the help of the Xtext involves the automatic creation of the corresponding *Ecore model* (i.e., a meta model of the EDL) that describes the structure of the EDL's *abstract syntax tree* (AST). The Xtext infers the Ecore model from the EDL grammar and adopts Ecore's EPackages to define the Ecore model. Ecore models are declared to be either inferred from the grammar or imported. By using specific directives, engineers instruct the Xtext to infer an EPackage from the grammar.

After the generation of the EDL meta-model (i.e., the Ecore model), we also get a set of tools and functionalities like the *parser*, the *linker*, the *type checker*, the *compiler* as well as editing support for Eclipse, IntelliJ IDEA and Web. The parser creates an in-memory object graph while experimenters define the script of each experiment. The object-graph is an instance of the EDL Ecore model. The parser is fed with a sequence of terminals and walks through the so-called *parser rules*. A parser rule produces a tree of non-terminal and terminal tokens i.e., the *parse tree*. Parser rules provide a building plan for the creation of EObjects that form the EDL semantic model (i.e., the AST). It should be noted that the EDL terminal rules are described using *Extended Backus-Naur Form*-like (EBNF) expressions.

15.5.2 The EDL Validator and Generator

The Xtext framework offers a set of automatic validation functionalities. Validation is very important to identify when the defined experiments are in 'agreement' with the EDL grammar. The first step of validation is performed by the available parser. The parser checks the syntactical correctness of any experiment while presenting error and warning messages. Such messages are automatically implied through the provided Xtext functionalities and show if an experiment complies with the terminology of the EDL grammar. In addition, the linker checks for broken cross-references between EDL concepts. The provided editors automatically validate all cross-links by navigating through the EDL model so that all the installed *Eclipse Modeling Framework* (EMF) proxies get resolved.

Apart from the automatic validation tools, RAWFIE EDL offers a set of custom tools adopted for validation purposes. The custom validator is written in the Xtend language[5] and adopts pure Java classes. The Xtend is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically, the Xtend has its roots in

[5]http://www.eclipse.org/xtend/

the Java programming language but is improved on many aspects. It offers extension methods for enhancing closed types with new functionalities while type inference and full support of generics offer compatibility with Java. Other advantages of the Xtend language are the operator overloading, powerful switch expressions, polymorphic method invocation, template expressions. The Xtend is very expressive, readable and any Xtend method can be invoked by Java classes in a transparent way.

The custom validator aims to define additional constraints for the defined experiments. In RAWFIE, the custom validator is adopted to define constraints in a semantic level for any experiment. The custom validator returns error or warning messages when violations in the experiment logic are present. The validator has access to the underlying database to get data related to the testbeds and UVs as well as to the experiments. Through this approach, RAWFIE platform can have full control of the defined experiments and forbid any action that cannot be performed by the nodes when the experiment will be realized. It should be noted that the custom validator is extended by adopting a Java class that includes the management of any check/functionality that is difficult to be handled by the Xtend language. In the following listing, we see a part of the validation script.

```
@Check
def checkDuration(RequirementsSection reqs) {
          if (Double.parseDouble(reqs.duration) <= 0)
        error("The experiment duration cannot be accepted!
              Please insert a positive number.", reqs,
              Literals.REQUIREMENTS_SECTION_DURATION, 101);
}
@Check
def checkMinDistance(RequirementsSection reqs) {
          if (Double.parseDouble(reqs.minDistance) <= 0)
        error("The experiment min distance cannot be accepted!
              Please insert a positive number.", reqs,
              Literals.REQUIREMENTS_SECTION_MIN_DISTANCE, 101);
}
@Check
def checkMaxDistance(RequirementsSection reqs) {
          if (Double.parseDouble(reqs.maxDistance) <= 0)
        error("The experiment max distance cannot be accepted!
                Please insert a positive number.", reqs,
                Literals.REQUIREMENTS_SECTION_MAX_DISTANCE, 101);
}
@Check
def checkAlgorithm(Algorithm users_algo) {
          if(!edlV.checkAlgorithm(users_algo.algName))
          error("Please type another algorithm. The " +
                users_algo.algName + " is not supported. Available
                algorithms: " + edlV.getAlgorithms(), users_algo,
                Literals.ALGORITHM_ALG_NAME, 101);
}
```

The Xtend language is also adopted for the creation of the EDL generator. The generator undertakes the responsibility of defining a set of files and code that will be executed directly by UVs. The generator is consisted of a set of Xtend files and multiple Java classes that depict each command defined by the experimenter into UVs commands. The Xtend can infer the types of variables, methods, closures, and so on and, thus, it can produce the mapping between EDL terminology and the target code.

15.5.3 The EDL Editors

The RAWFIE authoring tool offers two editors: the textual and the visual. Both editors offer their functionalities on top of the server part of the EDL. The server part is adopted to be the basis for building the Web version of the discussed editors. The EDL server is responsible to perform the validation (syntactic and semantic checking) as already described. All the backend Xtext functionalities are invoked with HTTP requests to the server-side component. The server immediately responds to any request and sends to the front end application data in the form of messages. The text content is either loaded from the Xtext server or provided through JavaScript. The Web integration of Xtext supports two operation modes: (i) *stateful mode* – in the stateful mode, an update request is sent to the server whenever the text content of the editor changes. With this approach a copy of the text is kept in the session state of the server, and many Xtext-related services such as AST parsing and validation are cached together with that copy; (ii) *stateless mode* – no update request is necessary when the text content changes, but the full text content is attached to all other service requests.

The client side of both editors is built through the adoption of JavaScript. A set of JavaScript files are responsible to visualize the proposed functionalities, accept experimenters commands and send the appropriate requests to the server-side component. The map presented in the visual editor is created with the adoption of OpenLayers[6]. OpenLayers is a pure JavaScript library for displaying map data in the most modern Web browsers, with no server-side dependencies. Experimenters have the opportunity to define in the graphical interface the routes and characteristics of the UxVs that they should perform during the execution of the experiment and, accordingly, the contents of both editors are synchronized. Hence, experimenters can easily switch from one editor to the other.

15.6 Case Study: Create and Launch an Experiment

In this case study, we show the steps required to define and launch an experiment. We assume that the experiment, initially, involves two (2) USV nodes. Assuming that the experimenter has booked the desired time for the experiment execution, he/she should login into the RAWFIE Web portal where he/she has access to the offered tools (Figure 15.3). There, the experimenter can access the authoring tool and use the provided editors. At the left, he/she can insert commands to the textual editor while at the right he/she can define nodes information in the visual editor.

For each editor, a set of buttons and menus are available. In Figure 15.3, we can see the available toolbars with a short description. Experimenters can use the available tools to insert the templates of specific commands. In Figure 15.4, we present an example where we insert the code templates for any basic part of an experiment.

In any step of the definition of an experiment, experimenter can use the provided content assist functionality to see the upcoming commands according to the EDL terminology (Figure 15.5). In addition, when an error is identified by the parser, the corresponding line of the experiment is marked with a red line and the error message is presented when the experimenter moves the mouse on the specific line (see Figure 15.6).

Nodes routes can be easily defined either in the textual or in the visual editor. As mentioned both editors are synchronized, thus, the experimenter can easily switch for the one to the other. In Figure 15.7, we see the routes for the two nodes under consideration. Experimenters can easily

[6]http://openlayers.org/

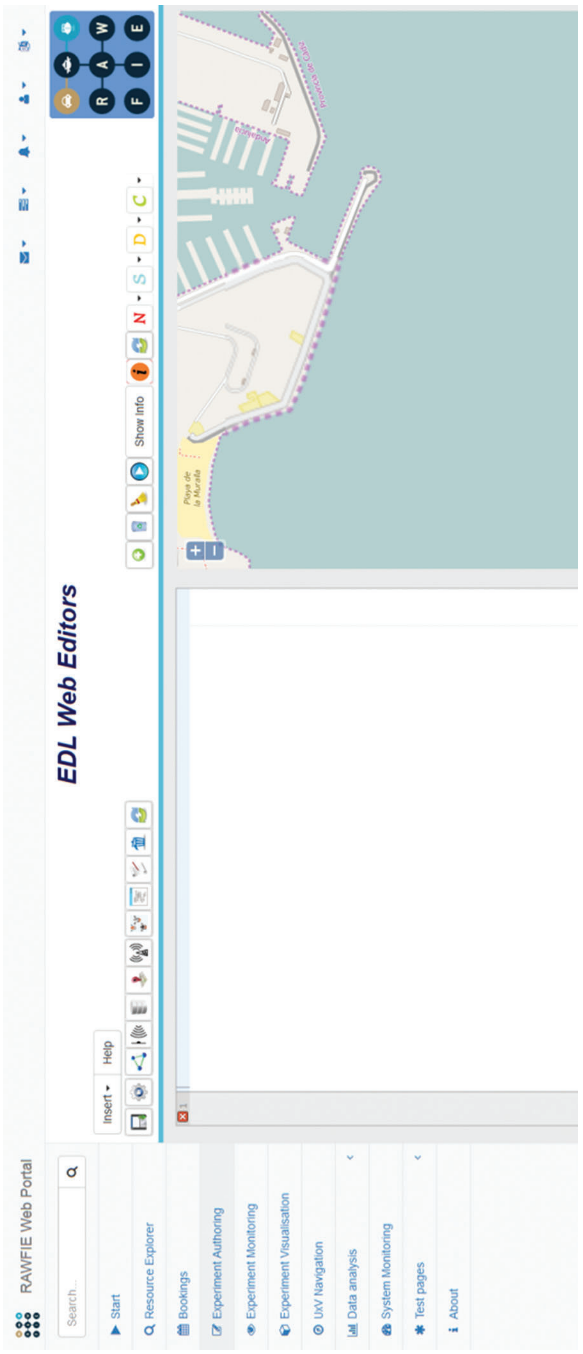


Figure 15.2 The RAWFIE Web portal and the authoring tool.

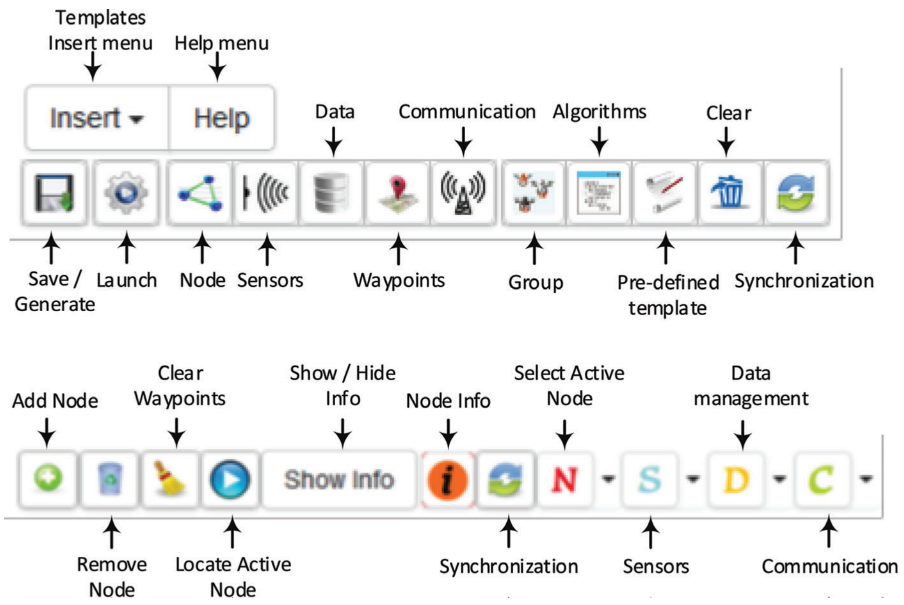
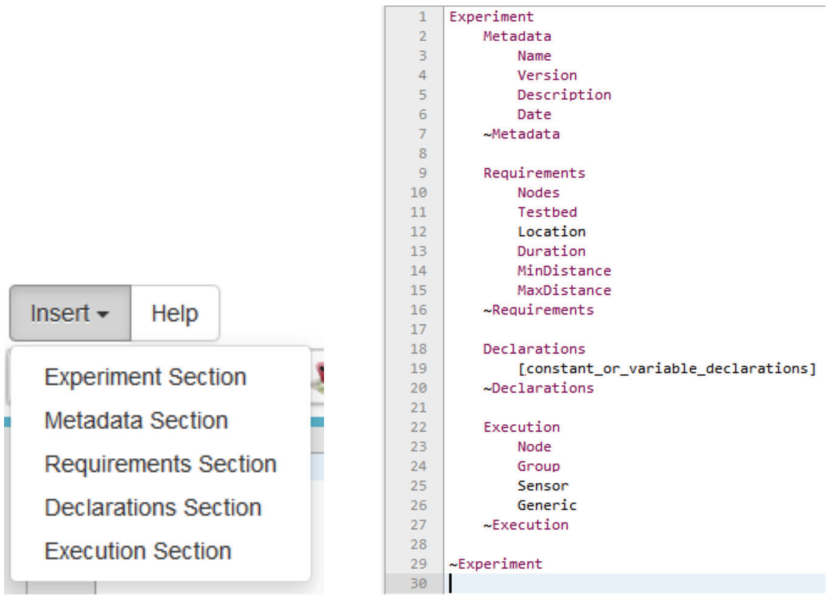
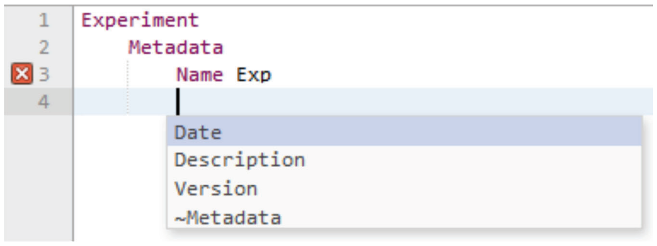


Figure 15.3 The editors' toolbars and buttons (above: the textual editor – below: the visual editor).



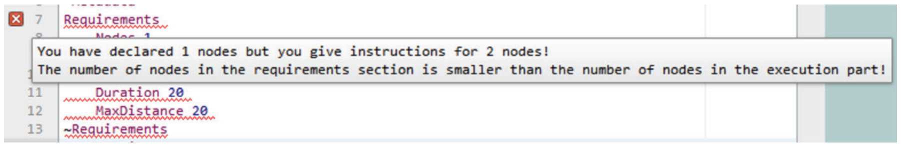
```
Experiment
    Metadata
        Name
        Version
        Description
        Date
    ~Metadata

    Requirements
        Nodes
        Testbed
        Location
        Duration
        MinDistance
        MaxDistance
    ~Requirements

    Declarations
        [constant_or_variable_declarations]
    ~Declarations

    Execution
        Node
        Group
        Sensor
        Generic
    ~Execution

~Experiment
```

Figure 15.4 An example of inserting code templates in the textual editor.



```
Experiment
    Metadata
        Name Exp
        |
```

Figure 15.5 The content assist functionality.



```
Requirements
You have declared 1 nodes but you give instructions for 2 nodes!
The number of nodes in the requirements section is smaller than the number of nodes in the execution part!
    Duration 20
    MaxDistance 20
~Requirements
```

Figure 15.6 Error identification by the parser.

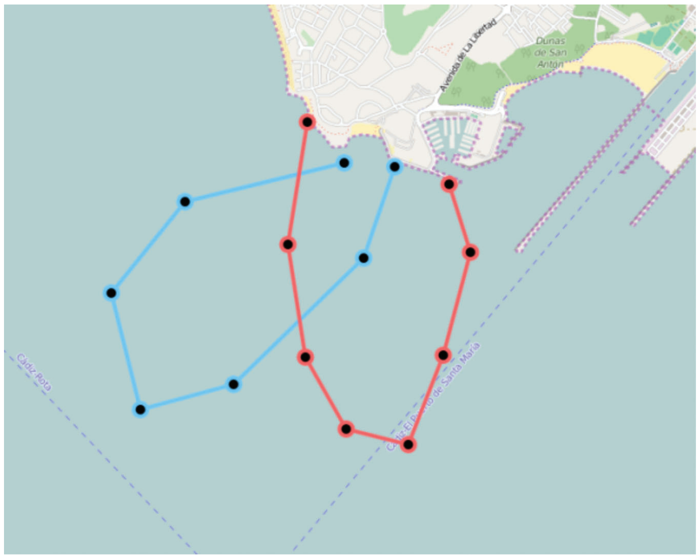


Figure 15.7 Waypoints definition for two USVs.

add more waypoints to each route by simply clinking on the map or they can move/remove waypoints by clicking on the mark (circle) of the waypoint that will be moved/eliminated. For performing any action with the route of a node, we should, first, select the corresponding layer as Figure 15.8 depicts.

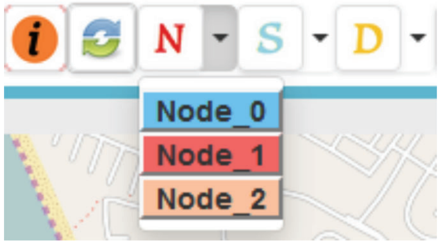


Figure 15.8 Node selection.

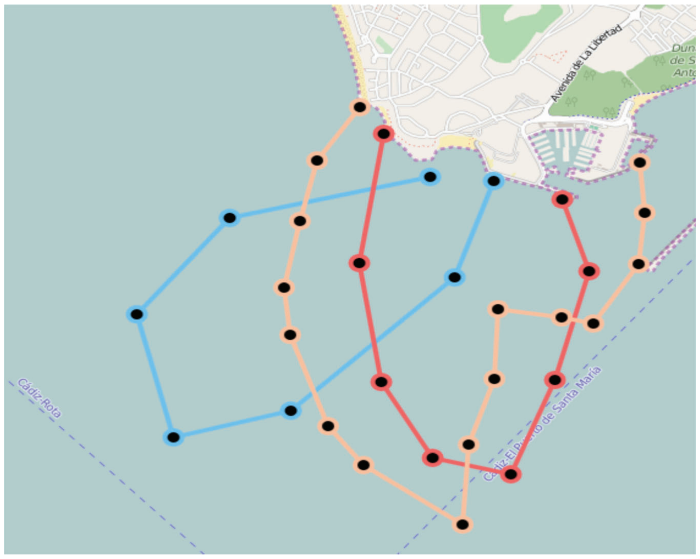
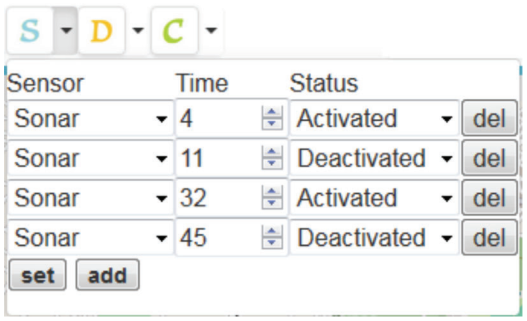


Figure 15.9 The addition of a node in the visual editor.

In Figure 15.9, we present the route of a third node that is added into our experiment.

For each node, we can also define the sensors, the data management algorithms or the communication interface that will be activated in specific time intervals during the execution of the experiment. In Figure 15.10, we present the popup window where the experimenter can manage the adoption of sensors for a USV. The specific example instructs a USV to activate the sonar from t = 4 to t = 10 and from t = 32 to t = 44. The same rationale stands for the invocation of data management algorithms and communication interfaces.



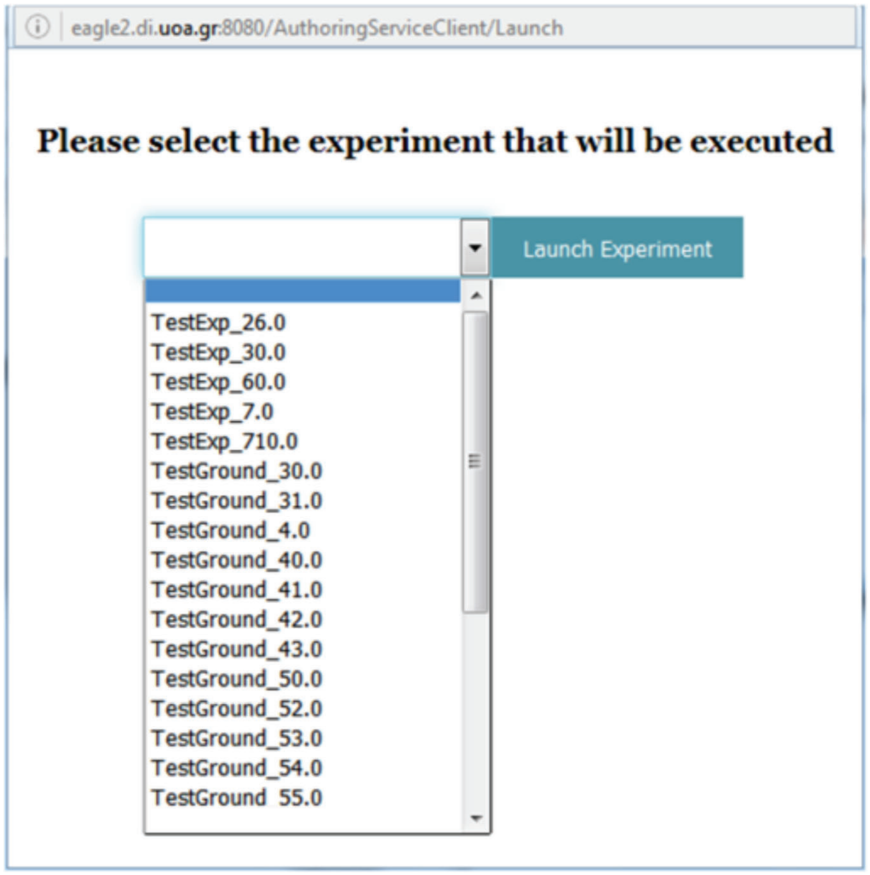
Sensor	Time	Status	
Sonar	4	Activated	del
Sonar	11	Deactivated	del
Sonar	32	Activated	del
Sonar	45	Deactivated	del

Figure 15.10 A part of the custom validation script.

After the definition of the experiment either in the textual or in the visual editor, experimenters can save the experiment by clicking on the corresponding save button (see Figure 15.3). At the same time, the appropriate files to be adopted by the remaining components of the RAWFIE architecture and the UxVs are generated. UxVs commands are stored in the database and, accordingly, can be adopted by the RAWFIE experiment controller component. The final step is to launch the experiment. Experimenters can press the corresponding button and a popup window is presented in the screen (Figure 15.11). Experimenters can select the experiment they desired by selecting the experiment ID from the drop down list. Just after the selection of the experiment, a call to the RAWFIE launching tool is realized and, thus, the experiment can be immediately executed.

15.7 Discussion and Future Extensions

The RAWFIE EDL offers the necessary conceptual basis for efficiently creating and launching experiments for mobile IoT applications. The provided editors incorporate all the appropriate functionalities to assist experts as well as non-experienced users to define their experiments in a user friendly environment. In the first place of future research/development plans is the incorporation of the error messaging mechanism in the visual editor. Hence, the visual editor will become the appropriate tool for building experiments while the textual part of the RAWFIE EDL will remain as the place where experimenters can insert generic information for their scripts. Such information is related with experiments metadata or requirements. The vision is to have a fully graphical interface and only information that is difficult to be inserted in the visual editor will remain as part of the textual editor. Hence,


eagle2.di.uoa.gr:8080/AuthoringServiceClient/Launch

Please select the experiment that will be executed

TestExp_26.0
TestExp_30.0
TestExp_60.0
TestExp_7.0
TestExp_710.0
TestGround_30.0
TestGround_31.0
TestGround_4.0
TestGround_40.0
TestGround_41.0
TestGround_42.0
TestGround_43.0
TestGround_50.0
TestGround_52.0
TestGround_53.0
TestGround_54.0
TestGround_55.0

Figure 15.11 Launching an experiment.

errors related to possible collisions, semantic/syntactic violations, etc., will be depicted in the visual part of the provided editors through the adoption of specific icons and colors. Experimenters will be immediately informed about the presence of an error accompanied by suggestions for fixing the error as it already stands for the textual editor.

In addition, another extension is to combine the RAWFIE authoring tool with the experiment monitoring mechanism to get insights on the experiment execution in real time. The aim is to have the system proposing possible modifications in the experiment logic and depict the part of the experiment that is currently executed. This way, experimenters can see in real time the experiment workflow as it is executed by the nodes and decide if it is possible to change specific aspects of the script. For instance, the authoring tool can be easily enhanced with functionalities related to the real time navigation of the UVs and, thus, to be fully aligned with experimenters needs. Specific toolbars can be provided for such purposes and experimenters will have the opportunity to produce/generate new commands during the execution of the experiment and UVs will have to change their routes/actions.

15.8 Conclusions

Mobile IoT applications can play an important role to the development of techniques/tools/services for improving people's lives in the new era of the IoT. This can be done through the adoption of mobile nodes interacting with their environment to collect and process data. Remote experimentation can build on top of such autonomous devices and become the means for experimenting with novel technologies before they are applied into real conditions. In addition, remote experimentation can become the basis for collecting and processing information related to many domains and, thus, to provide the means for creating or improving new applications. The research project RAWFIE offers a platform where numerous devices can be used in remote experimentation activities. Due to the complexity in defining instructions by adopting commands immediately executed by the autonomous devices, the use of a DLS is an easy way to define instructions at a high level. RAWFIE proposes a DSL called EDL that offers the necessary terminology for efficiently defining experiments. A validator and a generator are also proposed to validate the experiments and transform the high level commands into commands immediately executed by the devices. In this chapter, we describe the EDL, the validator, the generator and the available editors. We also elaborate on technical details and provide a case study where we create and launch an experiment from scratch. Our aim is to show the efficiency of the proposed approach while describing future extensions that will improve the offered functionalities and increase the satisfaction level of potential experimenters.

References

[1] Albu, M. M., Holbert, K. E., Heydt, G. T., Grigorescu, S. D., Trusca, V., 'Embedding remote experimentation in power engineering education', in *IEEE Transactions on Power Systems*, Vol. 19, No. 1, pp. 139–143, Feb. 2004.

[2] Ameixieira, C., Cardote, A., Neves, F., Meireles, R., Sargento, S., Coelho, L., Afonso, J., Areias, B., Mota, E., Costa, R. A., Matos, R., Barros, J., 'HarborNet: A Real-World Testbed for Vehicular Networks', CoRR abs/1312.1920, 2013.

[3] Arnold, B. R. T., van Deursen, A., Res, M., 'An algebraic specification of a language for describing financial products', In Martin Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, 1995, pp. 6–13.

[4] Baxter I., Pidgeon C., Mehlich M., 'DMS: Program transformation for practical scalable software evolution', In *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM Press, 2004, pp. 625–634.

[5] Blackwell, A., Britton, C., Cox, A., Green, T.R.G., Gurr, C., Kadoda, G., Kutar, M., Loomes, M., Nehaniv, C., Petre, M., Roast, C., Roe, C., Wong, A., Young, R., 'Cognitive dimensions of notations: Design tools for cognitive technology', In *Cognitive Technology: Instruments of Mind*, Springer-Verlag, 2001, pp. 325–341.

[6] Bonachea, D., Fisher, K., Rogers, A., Smith, F., 'Hancock: a language for processing very large-scale data', *SIGPLAN Notice*, vol. 35(1), 2000, pp. 163–176.

[7] de Lima, J. P. C., Rochadel, W., Silva, A. M., Simão, J. P. S., da Silva J. B., Alves, J. B. M., 'Application of remote experiments in basic education through mobile devices', *2014 IEEE Global Engineering Education Conference (EDUCON)*, Istanbul, 2014, pp. 1093–1096.

[8] Giordano, E., Tomatis, A., Ghosh, A., Pau, G., Gerla, M., 'C-VeT An Open Research Platform for VANETs: Evaluation of Peer to Peer Applications in Vehicular Networks', VTC Fall 2008: 1–2, 2008.

[9] Green, T. R. G., Blandford, A. E., Church, L., Roast, C. R., Clarke, S., 'Cognitive dimensions: achievements, new directions, and open questions', *Journal of Visual Languages & Computing*, vol. 17(4), 2006, pp. 328–365.

[10] Gupta, G., and Pontelli, E., 'Specification, Implementation, and Verification of Domain Specific Languages: A Logic Programming-Based Approach', Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I, 2002, pp. 211–239.

[11] Henriques P., Varanda Pereira M.J., Mernik M., Lenic M., Gray J., Wu H., 'Automatic generation of language-based tools using LISA', *IEE Proceedings Software*, vol. 152(2), 2005, pp. 54–69.

[12] Hermans, F., Pinzger, M., van Deursen, A., 'Domain-Specific Languages in Practice: A User Study on the Success Factors', Technical Report, Delft University of Technology, 2009.

[13] Herrera, O., Alves, G., Fuller, D., Aldunate, R., 'Remote Lab Experiments: Opening Possibilities for Distance Learning in Engineering Fields', chapter in Education for the 21st Century — Impact of ICT and Digital Resources, 2006.

[14] Hudak, P., 'Building domain-specific embedded languages', *ACM Computing Surveys*, vol. 28(4), 1996, pp. 196–202.

[15] Hudak, P., 'Modular domain specific languages and tools', In *Proceedings of the 5th International Conference on Software Reuse*, Washington, DC, USA, IEEE Computer Society, 1998.

[16] Hull, B., Bychkovsky, V., Zhang, Y., Chen, K., Goraczko, M., Miu, A., Shih, E., Balakrishnan, H., Madden, S., 'CarTel: a distributed mobile sensor computing system', SenSys: 125–138, 2006.

[17] Kelly, S., Tolvanen, J.-P., *'Domain-Specific Modeling Enabling Full Code Generation'*, John Wiley & Sons, Inc., 2008.

[18] Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., Walton, L., 'A software engineering experiment in software component generation', In *International Conference on Software Engineering*, 1996, pp. 542–552.

[19] Kolomvatsos, K., Valkanas, G., Hadjiefthymiades, S., 'Debugging Applications Created by a Domain Specific Language: The IPAC Case', *Elsevier Journal of Systems and Software (JSS)*, vol. 85(4), 2012, pp. 932–943.

[20] Kosar T., Martínez López P. E., Barrientos P. A., Mernik, M., 'A preliminary study on various implementation approaches of domain-specific language', *Information and Software Technology*, vol. 50(5), 2008, pp. 390–405.

[21] Kozil, T., Marek, S., Preparing and managing the remote experiment in education, in *Proc. of the 15th International Conference on Interactive Collaborative Learning (ICL)*, 2012.

[22] Mernik, M., Heering, J., Sloane, A. M., 'When and How to Develop Domain-Specific Languages', *ACM Computing Surveys (CSUR)*, vol. 37(4), 2005.

[23] Meta Object Facility http://www.omg.org/spec/MOF/

[24] Nordmann, A., Hochgeschwender, N., Wrede, S., 'A Survey on Domain-Specific Languages in Robotics', Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014, Bergamo, Italy, October 20–23, 2014.

[25] Oliveira, N., Pereira, M. J. V., Henriques, P. R., da Kruz, D., ‘Domain-Specific Languages: A Theoretical Survey’, Faculdade de Ciências da Universidade de Lisboa, 2009.

[26] Pereira, M. J. V., Mernik, M., da Cruz, D., Henriques, P. R., ‘Program comprehension for domain-specific languages’, Computer Science an Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications, vol. 5(2), 2008, pp. 1–17.

[27] Query/View/Transformation, http://www.omg.org/spec/QVT/

[28] Raymond James & Associates, ‘The Internet of Things – A Study in Hype, Reality, Disruption, and Growth’, online at http://www.vidyo.com/wp-content/uploads/The-Internet-of-Things-A-Study-in-Hype-Reality-Disruption-and-Growt....pdf, July 2016.

[29] Rebernak, D., Mernik, M., Wu, H., Gray, J., ‘Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars’, IET Software, vol. 3, Issue 3, 2009, pp. 184–200.

[30] Sadilek, D. A., and Wachsmuth, G., ‘Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages’, in *Proc. of the 4th European Conference on Model Driven Architecture: Foundations and Applications*, Berlin, Germany, 2008, pp. 63–78.

[31] SAFESPOT, available at: http://www.safespot-eu.org/

[32] Schmidt D., ‘Model-driven engineering’, IEEE Computer vol. 39(2), 2006, pp. 25–31.

[33] Spinellis, D., ‘Notable design patterns for domain-specific languages’, Journal of Systems and Software, vol. 56, 2001, pp. 91–99.

[34] Spinellis, D., Guruprasad, V., ‘Lightweight languages as software engineering tools’, In *Proceedings of the Conference on Domain-Specific Languages*, 1997, pp. 67–76.

[35] Sprinkle, J., Mernik, M., Tolvanen, J.-P., Spinellis, D., ‘What Kinds of Nails Need a Domain-Specific Hammer?’, *IEEE Software*, vol. 26(4), 2009, pp. 15–18.

[36] van Deursen, A., Klint, P., ‘Little languages: little maintenance’, *Journal of Software Maintenance*, vol. 10(2), 1998, pp. 75–92.

[37] van Deursen, A., Klint, P., Visser, J., ‘Domain-specific languages: an annotated bibliography’, *ACM SIGPLAN Notices*, vol. 35, 2000, pp. 26–36.

[38] Vermesan, O., Friess, P., Guillemin, P., Sundmaeker, H., Eisenhauer, M., Moessner, K., Le Gall, F., Cousin, P., ‘Internet of Things Strategic Research and Innovation Agenda’, in *Internet of Things – Converging Technologies for Smart Environments and Integrated Ecosystems*, River Publishers, 2013.

[39] Wu, H., Gray, J. and Mernik, M., 'Grammar-driven generation of domain-specific language debuggers', *Software Practice and Experience*, Vol. 38, 2008, pp. 1073–1103.

[40] Wu, H., Gray, J., and Mernik, M., 'Unit Testing for Domain-Specific Languages', IFIP Conference on DSLs, LNCS 5658, 2009, pp. 125–147.