A Tool for Modeling and Behavioral Tests Generation of Embedded Software ⋆

Thiago C. T. e Nascimento * **Lucas C. T. e Nascimento** ** **Rogerio A. de Carvalho** ***

* *Instituto Federal Fluminense, RJ, (e-mail: thiagocampostn@gmail.com)*
** *Instituto Federal Fluminense, RJ, (e-mail: lucascampostn@gmail.com)*
*** *Instituto Federal Fluminense, RJ, (e-mail: ratem@iff.edu.br)*

Abstract: This paper summarizes the functionalities of a module for the Modelio UML Case Tool named *WOOM Flow*. Its purpose is to extend the tool by allowing greater efficiency in the development of embedded systems, both in the design and the testing phases. This module interacts directly with Modelio's Finite State Machine (FSM) diagramming functionality, allowing the resulting diagram to be exported to the SCXML format. The model can then be used to produce a table that permits the assignment of responsibilities. This table gathers information that specifies the internal interactions of the system to be developed, which was previously specified through UML modelling of the behaviour expected by the system at a higher level. Technologies that focus on the structure, for example, Class-Responsibility-Collaboration Cards, suffer the possibility of losing information related to the system requirements during the transference to the coding phase. It happens because it directly associates a responsibility from a use case to a class. The advantage of using WOOM is to reduce as much as possible the loss of information by prioritizing the behaviour of the system. With the generated FSM and the assignments mapped by the table, it is possible to produce the skeleton of the embedded system, as well as the methods that will test it.

Keywords: finite state machines, embedded systems, automatic testing, software tool, system design

1. INTRODUCTION

At the end of the 1960s, with the increasing complexity of problems to be solved by computers, together with the increased demand for software, it was noticed that the techniques and methods for developing and expanding software were outdated and inefficient; thus causing the so-called "Software Crisis", which led to an increase in project costs, low-quality codes and an increase in bugs due to the great difficulty of maintaining the codes, according to Dijkstra (1972).

As noted by Hsu (2009), at the end of the 1990s the Object-Oriented Programming (OOP) paradigm, which emerged in the previous decade, became popular through proposals to solve the problems highlighted by the crisis, increasing efficiency, facilitating reuse, maintenance, extension, testing and documentation of codes. Snyder (1986) explains that OOP uses the concept of storing data in "objects" and no external agent can modify the object directly, thus requiring methods that interact with the object that causes its self-modification. This methodology creates decoupled and more descriptive codes, which are clearer and more specific.

Then, Test Driven Development (TDD) became popular in the early 2000s. According to Beck (2003); Janzen and Saiedian (2005), TDD is a software development process that follows the verification and validation procedures in a short cycle that ensures its expected functioning by encouraging the creation of multiple test cases according to the development progress, reducing duplication. According to Smart (2014); Solis and Wang (2011), Behaviour Driven Development (BDD) was developed by Dan North as a response to TDD. Comparatively, BDD promotes the development of tests that are structured similar to a written specification of the functionality, making their maintenance easier.

de Carvalho et al. (2013) explains the benefits of applying BDD in Information Systems projects by indicating that with the usage of BDD, implementation risks and effort are reduced. The WOOM Flow module makes usage of BDD, requiring the modelling of information systems in general, and embedded systems specifically, focusing on their behaviour. This approach ensures that a minimal (or potentially zero) amount of information related to system requirements is lost during the process since it keeps its structural aspects intact. Software modelling in UML using a CASE Tool, together with the rapid presentation of the relevant information of the software's structure and behaviour, has the potential to increase modelling

⋆ Fomented by the Programa Institucional de Bolsas de Iniciação em Desenvolvimento Tecnológico e Inovação from Conselho Nacional de Desenvolvimento Científico e Tecnológico.

productivity and testing, given that the module generates boilerplate code for testing each component.

2. BACKGROUND

There are main concepts and tools on which WOOM Flow depends heavily as its foundation due to their numerous advantages to the area of development of information and embedded systems. These benefits contribute to the formation of a clearer, more reliable and cohesive system that follows a modular design that aids collaboration and testability.

2.1 Unified Modeling Language

Established on OOP methods, Unified Modeling Language (UML), which emerged in the second half of the 1990s, is employed in the Software Engineering area to describe systems, according with Li and Chen (2009). It makes use of diagrams to represent structural and behavioural information about projects. With the application of UML, it is possible to visualize and analyze a design and find solutions in the early stages of development.

UML was designed as a way to make communication efficient around software production. In order to do this, it has several modalities in the generation of documents that enrich the development process, such as structural (classes, objects, components, implantation, packages and structure) and behavioural (use case, state machine, activities and interaction) diagrams.

2.2 Modelio

Modelio is a free software used to generate diagrams in the UML standard. Its specific application in this project is the construction of Finite State Machines with its modelling tools. Since Modelio is an open-source project with an active community and a wide range of documents, the decision to develop a module for the program would be more beneficial than alternatively creating a new, complete tool, which should reimplement multiple features that are already offered by Modelio. That way, duplication of effort is avoided by using a program that is already mature and maintained by an established community.

Given the scope of this project, the most useful resource is the tool for creating a "*State Machine diagram*". The software already comes with the ability to design ready-to-use state machines using a simple to use drag-and-drop interface. It's possible within the diagramming system to create different states, transitions and their respective conditions, which will then be parsed by the WOOM Flow.

Modelio only supports the distribution of the designed diagrams via image exporting. One of the purposes of the developed extension is to interact with the integrated FSM diagram models in order to export the diagrams into the SCXML standard. Therefore, the state machine can be stored into a file format that can be shared and interpreted by external tools, as well as being used to generate a WARC Table in a more user-friendly manner.

2.3 Finite State Machine

Wang (2019) describes a Finite State Machine (FSM) as a mathematical model of an abstract machine that stores information in different stages of execution called "states". Starting at the "initial state" or "start state", it can change from the current to next one through an operation called "transition". A transition can only occur if its condition has been fulfilled and/or through external input. Generally, the execution ends at a "final state" or "terminal state", where there are no additional transitions.

Usually, the representation of a FSM is made by circles that characterize different states of a system, additionally with arrows that denote transitions, and the conditions for these transitions to occur, as stated by Stallmann (1999); Pearce (2014). Starting at a specific state referred to as the initial state (usually represented by a black circle), the user can know the flow of the internal state of a system through a simple graphical representation.

It is possible to express a Finite State Machine in UML notation. Samek (2009) claims that the UML State Machine is more advanced than traditional state machines, due to its ability to divide similar behaviour between several states, which avoids the "explosion of complexity". Wagner et al. (2006) affirms this quality turns it into a powerful tool for describing the behaviour of hardware and software, as well as for testing them.

In order to make use of WOOM Flow, it is expected from the user the definition of a system that will be developed and the modelling of the different parts of the project as a Finite State Machine. For demonstration purposes, Figure 1 illustrates the operation of an FSM of a simplified crop irrigation system. This system will also be used to exemplify the tools offered by the module.

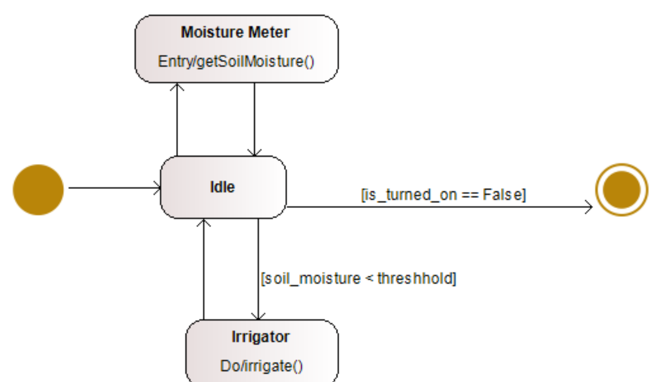

Figure 1. Example of a Finite State Machine created using Modelio.

Figure 1 presents an FSM with five states (including the initial and final states) and six transitions:

- `Initial State`: it's the point of the beginning of the FSM. It transitions to the *Idle* state with no conditions;
- `Idle`: the system's default state. It can transition to *Moisture Meter* state with no condition, to *Irrigator* state depending on a condition and to *Final State* when the system is turned off;

- `Moisture Meter`: entering the state, it calls a function that accesses a soil moisture sensor and stores the obtained information. When the operation finishes, it returns to the *Idle* state;
- `Irrigator`: using the value of soil moisture, the system transitions from *Idle* state to the *Irrigator* state when the moisture is lower than a threshold. When this happens, the system calls the "Irrigate" function that will activate the irrigation system. After it ends, the FSM transition back to *Idle* state;
- `Final State`: the state in which the system finishes working. The transition from *Idle* to *Final State* only happens if the variable "is_turned_off" is assigned to false, which means the user shut the system down.

This example denotes how a user can, through the application of diagramming, know all the internal states of the operation of a given system and the requirements to perform changes to it. In this way, these benefits can also be generally applied in software and hardware development as documents that seek to be informative and a mean to elaborate and improve the design of these systems during the development phase.

3. SCXML

SCXML or *State Chart XML* is a markup language. An XML extension, it is used to describe finite state machines in a standardized form. It was formally specified and launched by the W3C (or *World Wide Web Consortium*) in 2015 according to Rosenthal et al. (2015).

3.1 Usage

SCXML is used in this project as an intermediate form to store information about the overall structure of the system. Modelio does not offer the use of SCXML in a standardized way in its schematics. Instead, it makes use of a State Machine Diagram. Thus, there was an incentive to develop an extension that exports this diagram to a local file of extension *.scxml*.

This file can then be used subsequently as a basis for interaction between other systems. In this case, the file is read so that it can be integrated and managed in a WARC Table.

3.2 Example

Listing 1 below demonstrates an example of an exported code corresponding to the Finite State Machine in Figure 1.

Listing 1. Example of a generated .scxml file

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/
    scxml" version="1.0" name="State␣
    Machine1␣State␣Machine␣diagram">
    <state id="Idle">
        <transition event="Idle␣to␣
            Irrigator" cond="soil_moisture
            ␣<␣threshold" target="
            Irrigator" />
        <transition event="Idle␣to␣
            Moisture" cond="" target="
            Moisture␣Meter" />
        <transition event="Idle␣to␣Final"
             cond="is_turned_on␣==␣False"
            target="Final␣State" />
    </state>
    <initial id="Initial␣State">
        <transition event="Start␣to␣Idle"
             cond="" target="Idle" />
    </initial>
    <final id="Final␣State">
    </final>
    <state id="Irrigator">
        <onentry>
            <script>irrigate()</script>
        </onentry>
        <transition event="Irrigator␣to␣
            Idle" cond="" target="Idle" />
    </state>
    <state id="Moisture␣Meter">
        <onentry>
            <script>getSoilMoisture()</
                script>
        </onentry>
        <transition event="Moisture␣to␣
            Idle" cond="" target="Idle" />
    </state>
</scxml>
```

To describe the FSM, the following *tags* defined by SCXML standardization are used:

- `state`: defines the different states of an FSM. The *id* attribute states the title for the event;
- `initial`: indicates the first state of the system;
- `final`: indicates the last state of the system, where execution would end;
- `transition`: defines the transitions between different states of a system and composes the action that this system must apply before reaching the next state;
- `onentry`: composes the reaction that the system must apply when arriving at the new state.

4. MODULE

The module was developed for Modelio using its API and libraries such as the *Standard Widget Toolkit* (SWT), used to create a graphical interface that directly interacts with Modelio and its existing tools, expanding the software functionalities.

4.1 Background

de Carvalho and de Campos (2006) employs WOOM in their ERP5 project and they explain that despite having been created for generic applications, the method takes into consideration that the software platform supports an object-oriented language and a state-based workflow machine, which is compatible with that environment. Because of these considerations, the validity of the application of the method in the software here presented is based on the genericity of the method that allows compatibility with a UML State Machine.

WOOM Workflow, Object-Oriented Method (WOOM) is a method of modelling information systems in a structural format. It is described by de Carvalho and de Campos (2006) as a set of steps for modelling the structure and behaviour of an information system, using established UML and Contracts concepts. These steps include: identify process documents, identify classes, write use case, draw the state diagram and fill the WARC table.

WARC Table WARC Table (Workflow-Action-Responsible-Collaborators) is an artifact defined by de Carvalho (2005) that associates actions and reactions expected by the system.

According to de Carvalho and de Campos (2006), the WARC Table is used to associate structure to behaviour. Due to its prioritization on the process flow, there is a reduction in the loss of information during the system modelling phase, in addition to assisting in the execution and control of changes. The data displayed on the table allows the developer to simultaneously generate both the structure of the system to be developed (such as an embedded system) and the testing procedures of this system.

5. OPERATION

The program automates parts of WOOM and considers that certain steps have already been carried out beforehand. The changes take place as follows: it is assumed that both the identification of the process documents and the identification of classes have already occurred. It is also understood that the state diagram has already been drawn on the Modelio interface and, finally, it enables the contracts to be written on the user's account.

Based on these considerations, the detection of the classes and the design of the finite state machine must be built before the initialization of the program. With the completion of these steps, the module can be utilized. In it, classes can be inserted from the detections made previously; use cases are inferred as a consequence of the diagram and therefore do not need to be redefined. So, the user is left to fill the WARC table independently.

5.1 Usage

To use the module, it is necessary to open a Finite State Machine previously developed in Modelio. By using the design of the diagram, with its states and transitions constructed through the drag and click graphical interface, the module can be correctly executed. The project will appear similar to the one seen in Figure 1.

The program permits the user to export the created diagram in a SCXML file. The information from this file can be used by the software, by presenting it in the form of tables that can be edited by the user.

Several fields will be automatically filled in the tables, with information from the selected FSM. They can be adjusted in any way the user wants to reflect the architecture of the system being developed. Finally, all information can be stored in a file to be shared and uploaded by other users as a process document and tests can be generated.

5.2 Tabs

The module is divided into several tabs that allow the user to visualize and modify the information of the system that's being developed in a modularized format. These guides are used in the following workflow:

Home Shown in Figure 2, this tab is the extension's home screen, offering the user five buttons with the intention of saving and loading information.

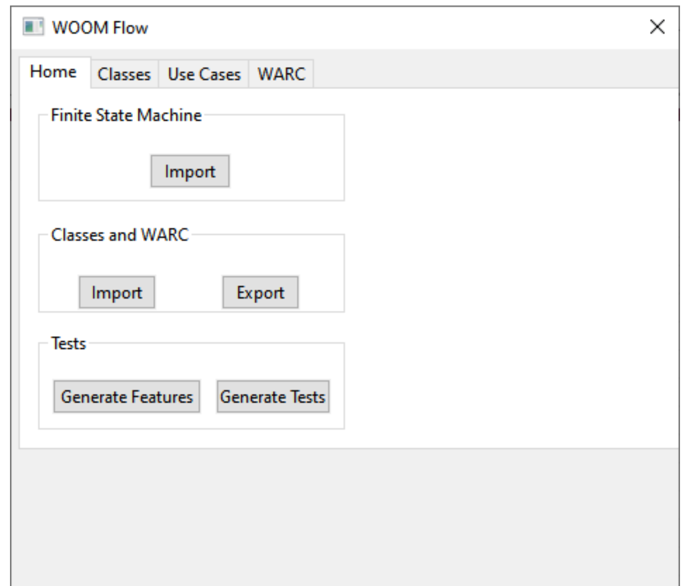

Figure 2. A screenshot of the program on the "Home" tab.

Under the "Finite State Machine" section:

- `Import`: opens a dialog for the user to select a "*SCXML*" state machine file. This data will then be used by the Use Cases and WARC tab.

Under the "Classes and WARC" section:

- `Export`: exports the data from the tabs "Classes" and "WARC" to a "*json*" file format, triggering a window dialogue where the user can choose the saving location and the file's name;
- `Import`: import the WOOM data previously stored on a "*json*" file format and populate the tables "Classes" and "WARC" in the extension with it.

Under the "Tests" section:

- `Generate feature`: uses the contents from the use cases in the WARC tab to generate a *.feature* file compatible with BDD development;
- `Generate tests`: generates a Python file with empty test cases for each of the use cases defined in the WARC tab. This serves the purpose of a starting point for testing the components being currently modelled;

Classes Displayed in Figure 3, it is an initially empty one-column table. The user can fill it by adding different classes. The user derives this information from the "Identify process documents" step of the WOOM process, intending to create classes that match the requirements of the system currently being developed. The contents of

this tab will be used in the WARC tab and can be later exported into a *.json* file for future reuse.

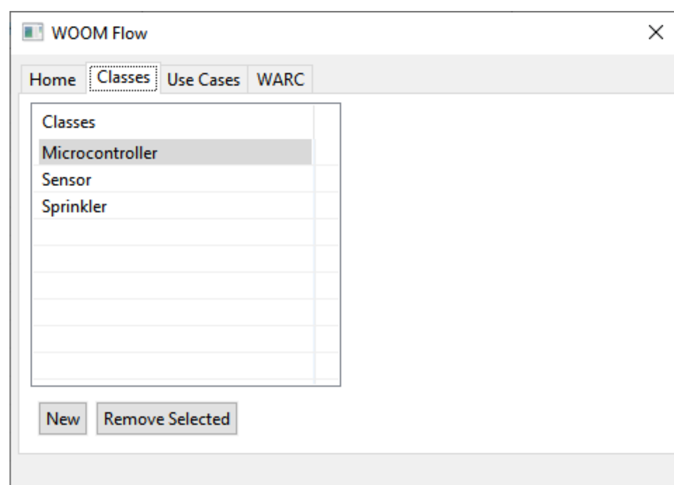

Figure 3. A screenshot of the program on the "Classes" tab.

Use Cases Illustrated in Figure 4, it is a two-column table in which the first includes all the actions specified in the diagram, and the following with the reactions associated with each one. It is automatically populated by the contents of the SCXML file by reading the states, their transitions, and internal attributes. This tab's purpose is to be an alternative method of visualizing the information provided by the FSM. This tab's function is purely for an alternative way of visualizing the information given from the FSM.

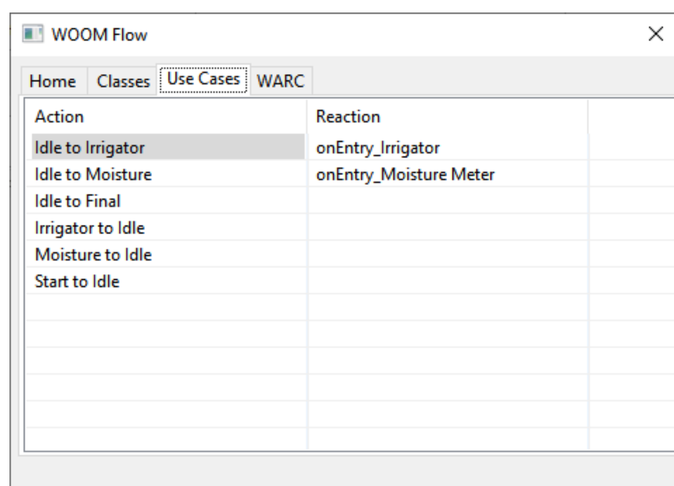

Figure 4. A screenshot of the program on the "Use Case" tab.

WARC Shown in Figure 5, it is a table with three columns: the first is labelled "Use Cases", populated with all actions and reactions on the "Use Case" tab; the second "Responsible" where the user can assign which of the classes specified in the "Classes" tab is responsible for the event (action or reaction), and the third "Collaborators", in which multiple classes can be designated that contribute to the operation of a given action or reaction of the system.

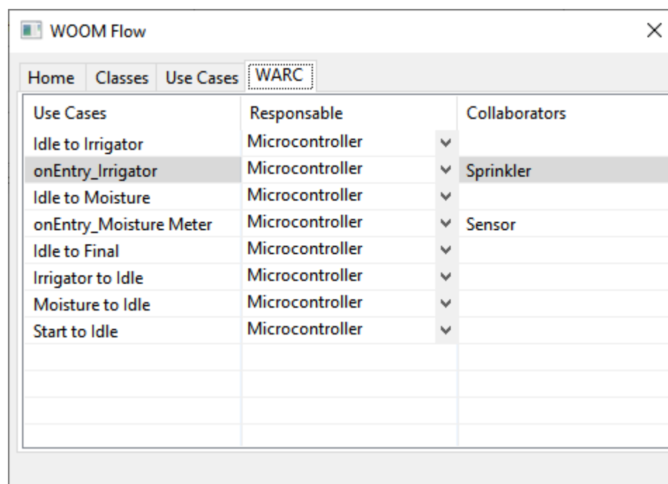

Figure 5. A screenshot of the program on the "WARC" tab.

6. TESTS

Given that tests are useful aggregation to the system development, they can be used to drive the effective development of a software, since the developer has the overview of its requirements, behaviours and expected responses.

This philosophy is compatible with the application of *Behavior Driven Development*. The information presented by the extension can be used by a developer in order to identify the behaviours that should be tested, just as the classes responsible by them and all their collaborators. This way, testing can be done at the same time the project is being built, which decreases the time demanded to identify and produce tests, while also serving as documentation for the expected behaviours.

6.1 Code Example

Listing 2 presents an example of a feature file for the *Idle to Irrigator*, *onEntry_Irrigator* and *onEntry_Moisture* use cases, using the Gherkin Syntax. This is automatically generated by using the information in the WARC Table, and it's expected that the user customizes them to fit the system.

Each use case receives a Scenario block in the feature file. In this example, the scenario *Idle to Irrigator* has as its responsible a class called "Microcontroller" and has only one collaborator named "Irrigator". In the "Given" segment, the data that will be used is identified and stored in the test case; in the "When" section, the event is run and the results of that action are stored; and finally, in the "Then" part the result is verified against the expected result. All other scenario blocks follows the same principle.

Listing 2. Example of a feature file for event testing

```
Feature: Test system behaviour
    Scenario: Idle to Irrigator
        Given the responsible "
            Microcontroller"
        When Irrigator is run
        Then the expected result is: "
            True"
```

```
Scenario: onEntry_Irrigator
    Given the responsible "
        Microcontroller"
    And the collaborator "Sprinkler"
    When onEntry_Irrigator is run
    Then the expected result is: "
        True"
Scenario: onEntry_Moisture Meter
    Given the responsible "
        Microcontroller"
    And the collaborator "Sensor"
    When onEntry_Moisture Meter is
        run
    Then the expected result is: "
        True"
```

7. CONCLUSION

The extension conceived for Modelio assembles UML modeling artifacts with the WARC table, efficiently supporting the development of embedded systems by explicitly presenting the information necessary for its elaboration. Another essential aspect for which this tool was conceived is in the development of tests cases for the project, since it is possible to readily visualize what are the current action and reaction events, which of them will be carried out by the system in different stages, which classes are responsible for every one of these events, and who are their collaborator classes.

Easy access to information, which is presented in an accessible format, is a remarkably important factor in the efficient elaboration of an embedded system. The application of agile methods such as BDD is encouraged in the development of tests that are used in every stage of the system so that it undergoes verification and validation.

Currently, the conversion of the elements from the diagrams to SCXML files only considers the information that is relevant for building a WARC Table. In the future, this conversion system can be made more versatile and embrace more complex behaviours such as state nesting. For this, the future task would be to implement the remaining of the SCXML standard as established by the W3C, and that can be mapped to UML elements. Another possible task would be better integration of SCXML as a file type that can produce a visual diagram in Modelio, instead of being used exclusively to generate the WARC Table. This would provide Modelio with an extensive SCXML capability for handling state machines.

ACKNOWLEDGEMENT

Thanks to the Instituto Federal Fluminense Campus Campos Centro and to the Polo de Inovação Campos dos Goytacazes.

REFERENCES

Beck, K. (2003). *Test-Driven Development: By Example.* Kent Beck signature book. Addison-Wesley. URL `https://books.google.com.br/books?id=gFgnde_vwMAC`.

de Carvalho, R.A. (2005). Device and method for information systems modeling. *Brazilian Patent PI0501998-2.*

de Carvalho, R.A. and de Campos, R. (2006). A development process proposal for the erp5 system. In *2006 IEEE International Conference on Systems, Man and Cybernetics*, volume 6, 4703–4708. IEEE. doi:10.1109/ICSMC.2006.385047.

de Carvalho, R.A., e Silva, F.L.d.C., Manhães, R.S., and de Oliveira, G.L. (2013). Implementing behavior driven development in an open source erp. In *Enterprise Information Systems of the Future*, 242–249. Springer. doi:10.1007/978-3-642-36611-6_22.

Dijkstra, E.W. (1972). The humble programmer. *Commun. ACM*, 15(10), 859–866. doi:10.1145/355604.361591.

Hsu, H. (2009). Connections between the software crisis and object-oriented programming.

Janzen, D. and Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9), 43–50. doi:10.1109/MC.2005.314.

Li, Q. and Chen, Y.L. (2009). Unified modeling language. In *Modeling and Analysis of Enterprise and Information Systems*, 209–224. Springer. doi:10.1007/978-3-540-89556-5_11.

Pearce, J. (2014). Finite State Machines. URL `http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/uml/fsm.htm`.

Rosenthal, N., Hosn, R., Bodell, M., Roxendal, J., Helbing, M., Auburn, R., Barnett, J., Carter, J., Burnett, D., Lager, T., Akolkar, R., Raman, T., McGlashan, S., and Reifenrath, K. (2015). State chart XML (SCXML): State machine notation for control abstraction. W3C recommendation, W3C. Https://www.w3.org/TR/2015/REC-scxml-20150901/.

Samek, M. (2009). Chapter 2 - a crash course in uml state machines. In M. Samek (ed.), *Practical UML Statecharts in C/C++ (Second Edition)*, 55–99. Newnes, Burlington, second edition edition. doi:10.1016/B978-0-7506-8706-5.00002-7. URL `https://www.sciencedirect.com/science/article/pii/B9780750687065000027`.

Smart, J. (2014). *BDD in Action: Behavior-Driven Development for the whole software lifecycle.* Manning Publications.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Not.*, 21(11), 38–45. doi:10.1145/960112.28702.

Solis, C. and Wang, X. (2011). A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, 383–387. doi:10.1109/SEAA.2011.76.

Stallmann, M. (1999). Finite-State Machines. URL `https://people.engr.ncsu.edu/efg/210/s99/Notes/fsm/`.

Wagner, F., Schmuki, R., Wagner, T., and Wolstenholme, P. (2006). *Modeling Software with Finite State Machines: A Practical Approach.* doi:10.1201/9781420013641.

Wang, J. (2019). *Formal Methods in Computer Science.* CRC Press. doi:10.1201/9780429184185. URL `https://books.google.com.br/books?id=OUyeDwAAQBAJ`.