Systematic DEVS Development: A Software Engineering Approach

Shafagh Jafer
Department of Electrical Computer Systems and Software Engineering
Embry-Riddle Aeronautical University, Daytona Beach, FL
jafers@erau.edu

Keywords: DEVS design, DEVS development, SDLC.

Abstract

This work attempts to present a software engineering approach towards DEVS development. By borrowing the Software Development Life Cycle (SDLC) concept, we map the five SDLC phases (i.e. Analysis, Design, Implementation, Testing, and Maintenance) to DEVS model construction. The idea is to introduce a systematic development approach that can be adapted by the DEVS community. Many researchers have proposed different strategies and tools to aid developing DEVS models. This work aims at standardizing this process by presenting a unified series of steps and phases that provide a model for the development and lifecycle management of a DEVS system. Aiming at focusing on the inherent model reusability feature of DEVS, we present an approach for the development, acquisition, and configuration of DEVS systems. This work serves as the "go-to" place for first-time DEVS developers.

1. INTRODUCTION

Component-based DEVS development has many attributes in common with object-oriented software development. They both require understanding the requirement specifications, then designing the overall architecture of the system, afterwards, providing the detailed specifications of system's components, then, implementing the system, testing its operation by validating and verifying each component as well as the overall system, and finally deploying it. These steps have been already studied and investigated by the software engineering community. Since its first introduction in 1960 [1], Software Development Life Cycle (SDLC) has been the most common development model used in software engineering. Advanced software engineering methods were developed since then to improve developer productivity by setting guidelines for activities to be performed at each software development phase. SDLC imposes the standard that defines all the tasks required for developing and maintaining a software product. These include a number of clearly defined and distinct work phases used by a software developer/engineer to plan for, design, implement, test, and deliver the software product to the customer.

SDLC defines two approaches to system development: *traditional approach*, and *object-oriented approach*. The first method, also referred to as structured system development [2], is a systems engineering and software engineering methodology for describing systems as a hierarchy of functions. With the introduction of high-level programming languages and the rapid advancements in software engineering tools and practices, the latter approach, object-oriented method, has achieved a lot of attention from systems designers and developers. Object-oriented (OO) techniques [3] view systems as collection of interacting objects working together to accomplish tasks. OO approach maps SDLC into analysis (OO Analysis), design (OO Design), and programming (OO Programming), allowing for defining and construing a complete software system based on collaborating objects.

Similar to OO techniques, the Discrete-Event System Specification (DEVS) [4] [6] allows defining an entire system through collaborative and interacting components. A DEVS-based system is composed of coupled (structural) and atomic (behavioral) entities. Atomic components can be viewed as objects communicating with each other by sending/receiving messages via ports. Given such architecture, it is straightforward to map SDLC to DEVS development. The purpose of this work is to demonstrate the one-to-one mapping of software development steps to DEVS modeling. We aim to demonstrate DEVS development within the SDLC framework, benefiting from advanced tools and techniques being introduced in software engineering every day. Concepts such as module-based design and implementation, component reuse, and unit and integrated testing can be easily adapted and well-mapped to DEVS modeling. This work takes a software engineering approach towards DEVS model development by presenting various OO techniques, with special focus on UML [7] technologies. The author has used this effort to teach third year software engineering students to develop DEVS systems as part of a Formal Modeling and Simulation undergraduate course.

The paper is organized as follows: Section 2 provides background information on SDLC concepts and techniques and introduces the DEVS theory. Section 3 provides a quick overview of existing DEVS development tools. Section 4 presents mapping of SDLC phases to DEVS development journey. A sample system is designed and implemented in

Section 5. Finally, Section 6 concludes the paper and summarizes the effort presented here.

2. BACKGROUND

2.1. SDLC

Software Development Life Cycle (SDLC) provides the overall framework for managing software development process. Throughout its development lifecycle, a software product goes through SDLC phases. SDLC defines five distinctive phases as shown in Figure 1:

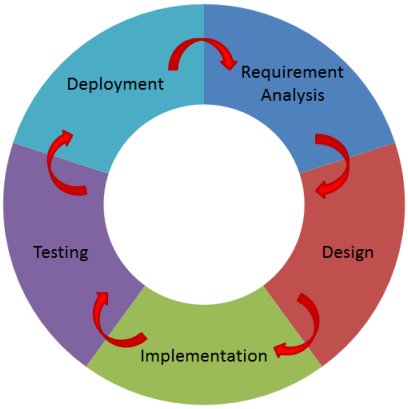

Figure 1. SDLC phases.

Requirement Analysis: This phase deals with gathering information to learn the problem domain, extracting data from system's description, and discovering requirements from the client. Once general requirements are gathered, an analysis of the scope of the project is conducted to clearly identify the set of functionalities that need to be developed. Requirements are then prioritized and alternatives are proposed and evaluated.

Design: The overall architecture of the software system is defined at this phase. High-level structure and detailed-level solutions are outlined based on requirements decisions made during the previous phase.

Implementation: The actual development takes place at this phase by constructing software components. Software engineers program the code according to the design given at the prior phase. As a result, a fully-functional software system is implemented at the end of this phase.

Testing: Validation and verification is performed at this phase to ensure correctness and quality of software system. Defects are recognized and removed, software units are tested individually and in an incrementally integrated fashion.

Deployment: The last SDLC phase starts directly after the code is properly tested, approved for release, and handled to client. This phase typically includes software installation, customization, maintenance, enhancement, and an extended period of user support.

SDLC is a repeating cycle since after the software is deployed, maintenance and additional support is required to address any undiscovered faults or requirements. In return this leads to possible redesign and updates of the software system. It is worth mentioning that phases are not always sequential; they can overlap, implying simultaneous activities being performed from two phases at the same time. Various software development models have been proposed to guide software engineers to carry on SDLC activities. Waterfall, spiral, incremental, prototyping, and iterative are examples of SDLC models [8]. The collection of software models, tools and techniques, referred to as *software engineering methodologies*, provide comprehensive guidelines to follow for completing every SDLC activity. Figure 2 illustrates the relationship among software methodologies components. In this paper we focus on tools and techniques for developing a software system. The following sections discuss software modeling techniques and Computer-Aided Software Engineering (CASE) tools.

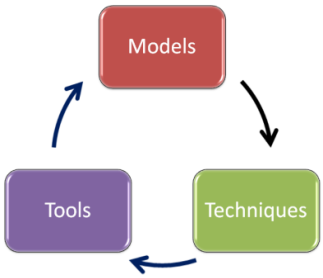

Figure 2. Software development methodology

2.1.1. SDLC Techniques

Among all object-oriented modeling techniques, the Unified Modeling Language (UML) [9] is the most commonly used specification. Proposed by the "three amigos" of the object-oriented world, James Rumbaugh, Grady Booch, and Ivar Jacobson, UML allows modeling an entire software system statically and dynamically. The various UML diagrams enable software engineers to model different aspects of the system through its SDLC phases. UML not only allows modeling the structure of the system, but also adds ability to describe the behavior of each object class/entity.

UML diagrams are mostly used during the Requirement, Design, and Implementation phase. UML Use Case diagram, implemented at the Requirement Analysis phase, greatly helps with requirement elicitation and analysis. It simply provides a representation of the system's interaction with outside world (i.e. system's users). Use cases focus on the "what" and not the "how" by providing a graphical representation of what the system must actually do.

UML diagrams that assist with the Design phase provide a high-level and a detailed-level representation of the system. The first high-level diagram used at the Design phase is the Domain Model. The Domain Model represents a conceptual model of all the entities related to a specific problem. In UML, a Class Diagram is used to represent the domain

model. The first Class diagram constructed at the Design phase, provides a high-level overview of the system's components. These components are discovered from the project description or the Needs Statement (from Requirement phase). The class diagram maps vocabulary and key concepts of the problem domain to classes, and identifies the relationships among them within the scope of the project. Towards the end of the Design phase, a more detailed class diagram is implemented, providing additional design decisions such as class attributes and methods associated with object-oriented models. Given the structural view of the problem domain in a class diagram, a UML Communication Diagram can help visualizing interaction among elements. On the other hand, a Sequence Diagram provides a more detailed view of objects interactions by also providing the order at which interactions take place. Both communication and sequence diagrams provide detailed design aspects that can be reflected back on the high-level class diagram to include much more design details.

In order to study and analyze the behavior of the system, UML provides the State Diagram. Towards the end of the Design phase, a state diagram helps understanding and verifying the behavior of the system. System's dynamics are translated into states and transitions depicting the internal behavior of the system, given different initial states and input values.

2.1.2. SDLC CASE Tools

Computer-Aided Software Engineering (CASE) tools are software tools designed to help software engineers complete development tasks. CASE tools assist software developers to organize and control the development of software, especially when projects are large, complex, and involve many software components and people. Various CASE tools have been produced for use at different stages of SDLC. A CASE tool may support developers through the requirement phase by serving as a repository of documents and program libraries containing the project's business plans, or allow constructing visual representation of the project's design, provide detailed code specification, assist with coding and testing, provide configuration management resources, and even support marketing and maintenance.

2.2. DEVS

Recent advances in computer technology have influenced modeling and simulation (M&S) techniques to become an effective approach for analyzing and designing a broad array of complex systems where a mathematical analysis is intractable. Just like software engineering SDLC, M&S takes similar approach in defining phases and activities to build and simulate a system. The simulation process begins with a problem to solve. First, the real system is observed (SDLC Requirements Analysis), its entities are identified (SDLC Design phase), and a model is constructed (SDLC Implementation phase). Then, the model is executed using a simulator consisting of a computer system, which executes the model's instructions and generates relevant output. These outputs are compared with the real system to verify the correctness of the model (SDLC Testing phase). Among the existing modeling and simulation techniques, the DEVS (Discrete Event System Specification) formalism [4] is regarded as one of the most developed general-purpose M&S frameworks for Discrete Event Dynamic Systems (DEDS). In DEVS, a real system is decomposed into behavioral (atomic) and structural (coupled) components. The system under study is modeled as a top coupled component encapsulating atomic or other internal coupled components. Components are linked through their input/output ports and interact with each other by sending/receiving event messages. Events can arrive at any time through input ports, but due to the discrete-event nature of DEVS, acceptable data can only be processed in a discrete fashion. The behavior of an atomic component is given by a state machine. Through their life time, atomic entities go through various states when transitions are triggered by incoming events. Figure 3 illustrates a DEVS system composed of one coupled (B) and two atomic (A and D) components.

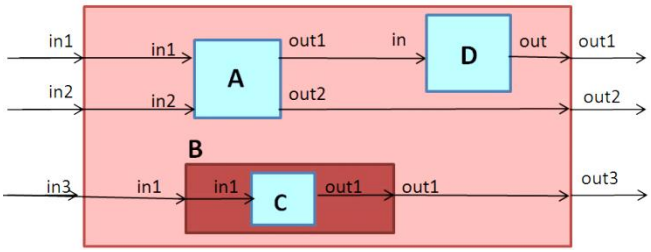

Figure 3. A DEVS system composed of atomic A and D, and coupled B components.

3. EXISTING DEVS TOOLS

Many DEVS tools have been proposed so far assisting DEVS developers in creating and implementing their DEVS systems. ADEVS, DEVS-C++, DEVS-Scheme, DEVS/CORBA, DEVS/HLA, DEVS/Grid, DEVSCluster, DEVSJAVA, JAMES, PyDEVS, PowerDEVS, SimBeans, DEVS/P2P, DEVS/RMI, and CD++ are some examples. A brief description of each of these tools is given in [13]. This paper takes an easy-to-follow approach to describe DEVS model development in CD++ toolkit. The steps discussed are tool-independent, meaning that they can be applied to any of the DEVS tools mentioned above. This paper serves as the "go-to" place for any beginner DEVS developer. The procedures discussed in the next section have been successfully used in an undergraduate course on Formal Modeling and Simulation with third year software engineering students. Students quickly understood and adapted the DEVS concept and developed individual DEVS projects using this approach. They fully comprehended the

DEVS theory and were able to map their initial system design to a completely functional DEVS product.

4. MAPPING SDLC TO DEVS DEVELOPMENT

Here we present how SDLC approach can be applied to construct and implement a DEVS system. We take a step-by-step approach to define DEVS development steps as they map to SDLC phases.

4.1. DEVS Requirements Analysis

As in any software development project, the first step towards DEVS modeling is gathering requirements and understanding the system functionalities. This phase starts with a needs statement or a system's description document. Given such document, the desired system is discussed and the "must-have" and "nice-to-have" functionalities are extracted. Such information is referred to as *functional* and *non-functional* requirements, which outline the structure, operational constraints, and the overall behavior of the system under study. It is also at this stage that the boundaries of the system are defined and specifications on how the system interacts with external environment are provided.

4.2. DEVS Design

With the boundaries of the system defined at the requirements phase, the Design starts with first sketching the overall conceptual design diagram. This is a high-level structural representation, visualizing the system boundaries and all sub-systems. The conceptual diagram can then be refined to include more details such as ports and connections among coupled and atomic components. Then, the DEVS coupled and atomic formalism specifications are defined, detailing information about input and output ports, as well as ports and components couplings. The atomic formalism specification would also include details regarding states and their duration, internal and external transition behavior, and output generation.

Then, a state diagram is created for each atomic component to clearly illustrate its various states, state transitions, input events triggering state change, output events generated at the end of the states, and states time durations.

4.3. DEVS Implementation

This phase involves implementing the internal behavior of the DEVS atomic components complying with the system's conceptual design and the state diagrams. Each state diagram is translated into a separate atomic DEVS implementation. According to the atomic component's state diagram, the algorithms for internal and external transitions are first implemented. Then, output generation and state duration are implemented. A structural script is written to define the system's decomposition and connections among coupled and atomic components.

A DEVS development environment is used at this phase to assist developers to easily implement their system's details. For demonstration purposes, the CD++ toolkit [5] was used to implement the sample model defined in this paper (to be discussed in Section 5).

4.4. DEVS Testing

DEVS testing is conducted by analyzing events handling, output generation, and states timing. Each atomic component is tested individually (Unit Testing) by injecting events at different times and observing the behavior of the component in response to events arrivals. The state change and output generation timing are then compared against the model's specification (defined on the state diagram), confirming test acceptance or rejection. With compliance to the system's conceptual design, atomic and coupled components are incrementally integrated to perform Integrated and System testing. Acceptance of all unit and integrated tests yield the overall system acceptance concluding the test phase.

4.5. DEVS Deployment

A fully functional and tested DEVS system is ready for deployment. With its component-based infrastructure, DEVS provides an ideal solution to model-continuity and component-reusability. DEVS models can be easily integrated and extended to accommodate larger Systems developments. Additional functionalities and behavior can also be easily added to existing models.

5. SAMPLE MODELS

This section describes developing a simplified ATM machine system with CASE (UML and java) and DEVS (CD++ tool). The goal is to illustrate one-to-one mapping of software engineering SDLC to DEVS development by taking a step-by-step approach where the same model is developed using the two technologies. This section provides an excellent guide to instructors and first-time DEVS developers to understand how a DEVS system is modeled and implemented. The system description is provided first. Then the five SDLC phases are implemented under both CASE and DEVS.

5.1. System Specification

An ATM Machine consists of a User Interface (UI) and a Cash Dispenser (CD) unit. A user interacts with the system through the UI by swiping their bank card. The system only allows cash withdrawal. Once the user's card is in, the cash amount to be withdrawn must be entered via the UI. The system will forward the amount information to the CD unit. The CD verifies that the amount requested is within acceptable limit and dispenses the cash.

The following functional requirements are to be addressed:

- It takes the UI exactly one minute to verify card information (assume all cards entered into the system are verifiable);
- Once the card is verified, the user has five minutes to enter the withdraw amount. If the amount is not entered

within five minutes, the UI sends out an error message and the ATM machine goes back to *Idle*;

- The CD handles the dispense action in exactly one minute, and then dispenses the requested amount.

5.2. System Requirements Analysis

Based on the system's description, the functional requirements, and the assumptions provided, the following functionalities are identified:

- User interaction with the system through UI
- Cash withdrawal request submitted by user through UI
- System dispensing cash through CD
- Timing constraints on card verification, arrival of withdrawal amount request, and cash dispensing activity

To outline the boundaries of the system, a UML Use Case diagram is created as in Figure 4.

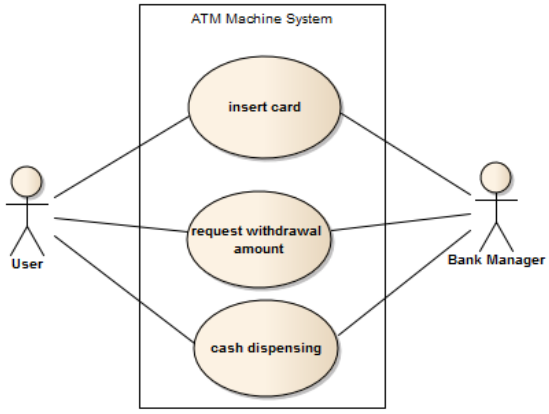

Figure 4. ATM system Use Case diagram.

5.3. System Design

The ATM system design starts with a conceptual diagram outlining system decomposition. Figure 5 illustrates the two entities of the ATM machine. This diagram serves as the high-level design diagram which will be refined and revisited differently in CASE and DEVS development approach.

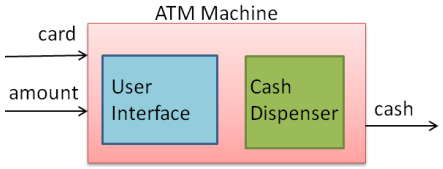

Figure 5. System's conceptual design.

5.3.1. CASE high-level and low-level design

Based on the conceptual design of the system, with the aid of a CASE tool (Enterprise Architect by SparxSystems [10]) and UML, a Class Diagram is created serving as the high-level design. The first version of the class diagram only demonstrates the System structure composing of classes and their relationships. This class diagram will be incrementally refined adding more details such as attributes and methods encapsulated by each class. Low-level design starts by creating communication and sequence diagrams to present objects interactions. Such diagrams will be used to add details to the original class diagram. Figure 6 demonstrates the detailed low-level class diagram.

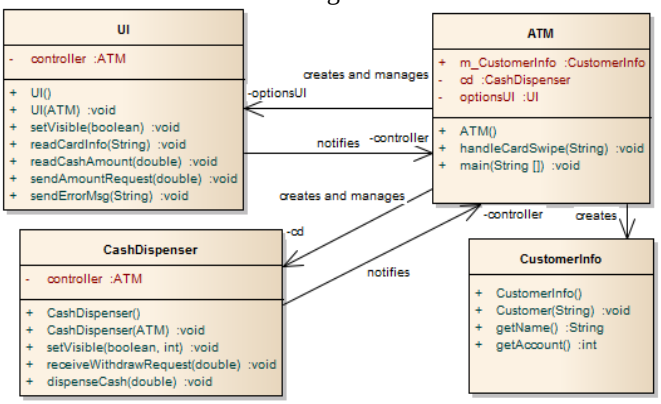

Figure 6. System Class diagram.

At this stage, a State diagram is generated to model the behavior of the controller (main processing unit of the system, i.e. the ATM Class). The behavior presented by the State diagram illustrated in **Figure 7** is translated into actual source code during the implementation phase.

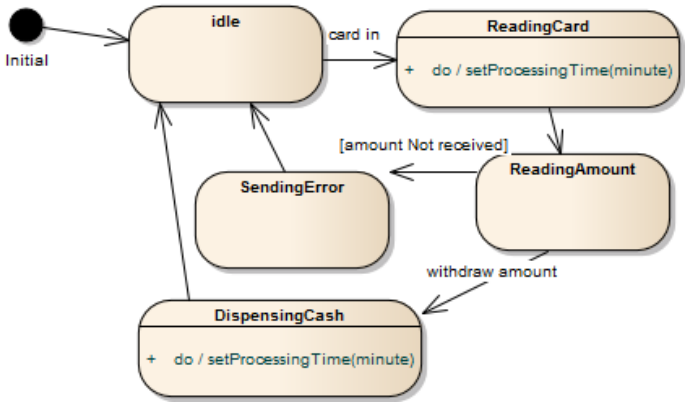

Figure 7. System State diagram.

5.3.2 DEVS HIGH-LEVEL AND LOW-LEVEL DESIGN

As illustrated in Figure 8, first the System's Block Diagram (Conceptual Model) is created defining all coupled and atomic components and their connections (input/output ports linkage).

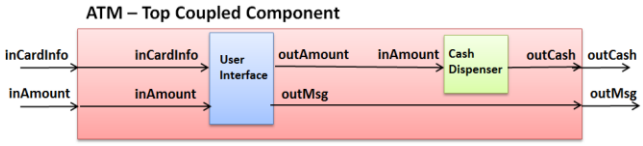

Figure 8. DEVS system Block diagram.

The next step is writing the formalism definitions for all the coupled and atomic components as follows [4]:

ATMTop = <X, Y, D, M_d, EIC, EOC, IC, Select>
X = {(inCardInfo, 1...3), (inAmount, 1...400)}
Y = {(outCash, 1...400), (outMsg, 0)}
D = {UI, CD}
M_d = {M_{UI}, M_{CD} }
EIC = {((Top, inCardInfo), (UI, inCardInfo)), ((Top, inAmount), (UI, inAmount))}
EOC = {((Top, outCash), (CD, outCash)), ((Top, outMsg), (UI, outMsg))}
IC = {(UI, outAmount), (CD, inAmount)}
Select = {CD, UI}

The next design diagram is the detailed modeling of the atomic components' behaviors. To get this step done, a state diagram needs to be created for each atomic component. Note that state diagrams cannot be generated for coupled components as coupled DEVS only defines the hierarchy and structural decomposition of the DEVS system. The User Interface (UI) state diagram is presented in Figure 9.

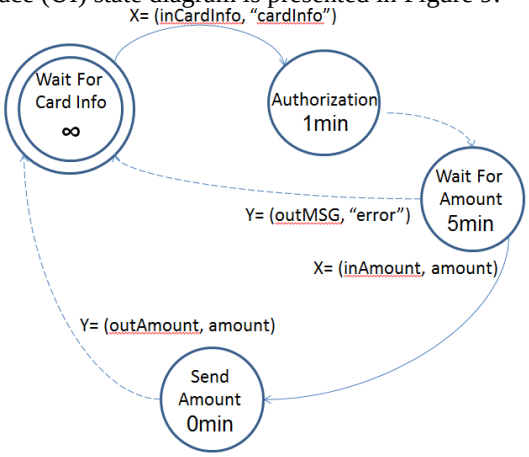

Figure 9. UI State diagram.

At the beginning the UI is in state *waitForCardInfo*. It remains in this state until an input event arrives through its *inCardInfo* with a value of *cardInfo*. Arrival of this event will trigger transition to *Authorization* state. The UI is in this state for exactly one minute (time required to process card information), after which it transits to *WaitForAmount*. While in this state, if an input arrives from *inAmount* port holding the requested withdrawal amount within five minutes, the UI accepts the input and transits to a new state (*SendAmount*) to dispense the cash. However, if the UI did not receive an input within the specified time frame of five minutes, an error message is generated as an output event and sent out through *outMsg* port, and then the UI transits to *WaitForCardInfo* and the cycle is repeated over. On the other hand, when the UI is in *SendAmount* state because the withdrawal request arrived within the correct time frame, immediately an output is generated depicting cash dispense, and a state transition to *WaitForCardInfo* occurs. Having the state diagram modeled, it is then possible to provide the UI DEVS definition as following:

UI = <X, Y, S, δ_{int}, δ_{ext}, λ, ta>
X = {(inCardInfo, 1…3), (inAmount, 1…400)}
Y = {(outAmount, 1…400), (outMsg, 0)}
S = {*WaitForCardInfo, Authorization, WaitForAmount, SendAmount*}
δ_{int} (S -> S'){ //only cares for states that have **finite** *ta*
if (S = *Authorization*) -> *S = WaitForAmount*
If (s = WaitForAmount) -> s = WaitForCardInfo
if (S = *SendAmount*) -> *S = WaitForCardInfo*
}
λ (S -> Y) { //we only have output for states that have **finite** *ta*
if (S = *WaitForAmount*) -> *out(outMsg, 0)*
if (S = *SendAmount*) -> *out(outAmount, 0…400)*
}
δ_{ext} (S,e,X -> S'){ //only cares for states that can receive input
if (S == *WaitForCardInfo && e = anytime* && X == (inCardInfo,1…3)) -> *S = Authorization*
if (S == *WaitForAmount && e < 5* && X == (inAmount, 1…400)) -> *S = SendAmount*
}
ta (S -> R^+) {
WaitForCardInfo -> ∞, *Authorization* -> *1, WaitForAmount* -> 5 , *SendAmount* -> *0*}

Similarly, a state diagram is defined for the *CashDispenser* (CD) atomic component (refer to Figure 10).

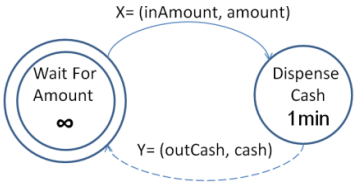

Figure 10. Cash Dispenser State diagram.

The CD behavior is very simple. There are only two states: *WaitForAmount* and *DispenseCash*. Initially the CD is in state *WaitForAmount*. It remains in this state until an input event arrives at *inAmount* port holding the withdrawal amount requested, causing a transition to the *DispenseCash* state. This state is timed, meaning that, when the state duration expires, an output event is generated representing the cash withdrawal action through the ATM machine. The CD DEVS definition is then given according to its state diagram as follows:

CD = <X, Y, S, δ_{int}, δ_{ext}, λ, ta>
X = { (inAmount, 1…400)}
Y = {(cashOut, 1…400)}
S = {*WaitForAmount, DispenseCash*}
δ_{int} (S -> S'){ //only cares for states that have **finite** *ta*
if (S = *DispenseCash*) -> *S = WaitForAmount*
}
λ (S -> Y) { //we only have output for states that have **finite** *ta*

```
        if (S = DispenseCash) -> out(outCash, 1)
        }
δext (S,e,X -> S'){    //only cares for states that can receive input
        if (S == WaitForAmount && e = anytime && X == (inAmount,1…400))
          -> S = DispenseCash
        }
ta (S -> R+) {
        WaitForAmount -> ∞,  DispenseCash -> 1 min}
```

5.4. System Implementation

5.4.1. CASE Implementation

One of the approaches to start the implementation phase in a CASE project is to use the Forward Engineering process where using a CASE tool, the initial code skeleton of the system is automatically generated. Enterprise Architect provides this mechanism where with a click of a button, skeletons for all the classes with their attributes and methods, and their relationships are implemented in an easy and quick step. Once the source code initial skeleton is generated, it is then a straight forward process to translate the state diagrams behavior into lines of code using an integrated development environment like Eclipse Java. The implementation phase takes a good amount of time as the core development takes place during this stage. Due to space limitations, code snippets are not presented in the paper.

5.4.2. DEVS Implementation

With the coupled and atomic components definitions, and the state diagrams at hand, the next step is to choose a DEVS modeling tool. The CD++ toolkit provides an easy-to-use framework to implement DEVS models in C++. First a model file (ATM.MA) is implemented defining the top ATM coupled component and the decomposition of the system into the two internal atomic entities. The ATM system model file is defined as follows:

```
[top]
components : CD@CashDispenser
UI@UserInterface
out : outCash outMsg
in : inCardInfo inAmount
Link : inCardInfo inCardInfo@UI
Link : inAmount inAmount@UI
Link : outAmount@UI inAmount@CD
Link : outMsg@UI outMsg
Link : outCash@CD outCash
```

Next, a ".cpp" and a corresponding ".h" file are created for every atomic component to implement its behavior. The CD++ tool provides a template for these two files, making it a very simple and quick step. The ".h" file includes definition of that atomic component's states, the time duration for each state, and declarations for the four core DEVS methods: initial, internal transition, external transition, and output function. A DEVS developer provides the implementation for these functions by mapping the DEVS atomic definition and behavior (as defined on the state diagram) to source code. The UI code snippet mapping the state diagram algorithm is presented in Figure 11 and Figure 12.

```
class UserInterface : public Atomic{
...
protected:
        Model &initFunction();
        Model &externalFunction( const
        ExternalMessage & );
        Model &internalFunction( const
        InternalMessage & );
        Model &outputFunction( const
        InternalMessage & );
private:
        const Port &inCardInfo, &inAmount;
        ...
        Time waitTime5;
        int amount;
        enum State{ WaitForCardInfo,Authorization,
        WaitForAmount,SendAmount};
                ...
```

Figure 11. Implementing "UI.h" file.

```
Model &UserInterface::externalFunction( const
ExternalMessage &msg ){
 if ((state == WaitForCardInfo) && (msg.port() ==
     inCardInfo) && … {
     state = Authorization;
     //setting state duration
     holdIn(Atomic::active, waitTime1);
    }
 else if ((state == WaitForAmount) && (msg.port()
      == inAmount) && … {
      state = SendAmount;
      holdIn(Atomic::active, waitTime0);
    }
return *this;
}
Model &UserInterface::internalFunction( const
InternalMessage & ){
 switch(state){
 case Authorization:
      state = WaitForAmount;
      holdIn(Atomic::active, waitTime5);
 break;
 case WaitForAmount:
      //state with no expiration
      state = WaitForCardInfo; passivate();
 break;
 case SendAmount:
      state = WaitForCardInfo; passivate();
 break; };
return *this;
}
```

Figure 12. Implementing "UI.cpp" file.

5.5. System Testing

"Trying to improve the quality of software by doing more testing is like trying to lose weight by weighing yourself more often" [11]. Software testing represents the ultimate review of the requirements specification, the design, and the code. Quality software must be free of defects or as close to it as possible. Software quality is typically assessed through Validation (*did we build the right product?*), and Verification (*did we build the product right?*). Testing is the most widely used approach to manage software quality. Through testing, we try to prove a program is correct and uncover any undiscovered defects. Many organizations spend 40-50% of development time in testing.

5.5.1. CASE Testing

To date, many software testing strategies have been proposed. The two general categories of testing are: Unit Testing, and Integrated Testing. Unit testing checks that an individual program unit (subprogram, object, package, module) behaves correctly. On the other hand, the integrated testing ensures that when program modules are incrementally integrated they would still perform as expected. Many CASE tools and techniques have been introduced to assist software developers conduct unit and integrated testing. Among these, the java unit testing framework, *JUnit* [12]has been widely used among Java developers. It allows developers to write and run repeatable tests. Built into Eclipse IDE, it automatically calls methods and compares the results they return against expected results. *Junit* can be easily used to test each class of the ATM system.

5.5.2. DEVS Testing

DEVS unit testing can be performed by analyzing the behavior of each atomic unit separately. For this purpose, an atomic component is injected with various inputs and timing sets, and the corresponding outputs and their timing are verified against the model specification as outlined on the state diagram. Let's take the Cash Dispenser atomic component and apply unit testing to it. First an event file is created specifying the value, the port, and the time at which an input event is injected into the CD component. Figure 13 illustrates the content of the event file (".ev"). The event file has the following format:

eventTime	*port_name*	*eventValue*
00:01:00:000	inAmount	100

Figure 13. "in.ev" file for testing the Cash Dispenser unit

Given this input, from the CD state diagram, the correctness of the unit can be verified when a $100 cash value is withdrawn through the CD outAmount port at time 00:02:00:000 (it takes the CD unit one minute to process a withdraw event). The output file demonstrated in Figure 14 shows the result of the unit test written to a ".out" file.

00:02:00:000	outAmount	100

Figure 14. "out.out" file generated at the end of Cash Dispenser unit test

Once all atomic units are tested separately, then their corresponding coupled components are tested (incremental integrated testing). By grouping integrated tests, finally an overall system integrated test can be conducted verifying that the system works as a whole.

5.6. System Deployment

DEVS unit and coupled components can be easily reused and expanded to allow continuous system improvement and adaptivity. With its component-oriented and model-driven architecture, any DEVS system can be reused to build other systems, making DEVS a powerful technology that can be easily maintained and adapted for complex systems development. A large model repository database is available in where many DEVS models have been implemented in various fields and topics [6].

6. CONCLUSION

A software engineering approach is taken to present the DEVS theory and teach DEVS development. This work presented a DEVS development endeavor in the framework of Software Development Life Cycle. Each SDLC phase is mapped to a DEVS construction stage to allow usage of software engineering concepts and practices. The procedure presented here was in fact applied to teach first-time DEVS developer group of students in their third year of a software engineering curriculum. DEVS theory and development process was easily picked up by students and fully functional DEVS systems were built using the CD++ toolkit. The effort presented here serves as the "go-to" place to introduce DEVS modeling and development to first-time DEVS users.

7. REFERENCE

[1] Elliott, G., Strachan, J. "Global Business Information Technology". 2004.

[2] Davis, W. "Tools and Techniques for Structured Systems Analysis and Design". Addison-Wesley. ISBN 0-201-10274-9. 1992.

[3] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. "Object-Oriented Modeling and Design". Prentice-Hall, Englewood Cliffs, NJ, 1991.

[4] Zeigler, B., Praehofer, P. H., and Kim, T. G. "Theory of Modeling and Simulation". Academic Press. 2000.

[5] G. Wainer, "CD++: A Toolkit to Develop DEVS Models", Software – Practice and Experience, 32(13), pp. 1261-1306, 2002.

[6] http://www.sce.carleton.ca/faculty/wainer/wbgraf/doku.php?id=model_samples:start. [Last accessed: February 2013]

[7] Fowler, M. "UML Distilled: A Brief Guide to the Standard Object Modeling Language". Addison-Wesley Professional. 2004.

[8] Pressman, R. "Software Engineering: A Practitioner's Approach". McGraw-Hill Inc., New York, NY. 2007.

[9] Jacobson, I., Booch, G., Rumbaugh, J. "The unified software development process". Addison-Wesley Longman Publishing Co., Inc., Boston, MA. 1999.

[10] Sparx Systems. Available at: http://www.sparxsystems.com. [Last accessed: October 2013]

[11] McConnell, S. "Code complete". O'Reilly Media, Inc., 2004.

[12] Massol, V. "Junit in action". Dreamtech Press, 2004.

[13] Jafer, S. "New Algorithms for the Parallel CD++ Simulation Environment". M.A.Sc Thesis, Carleton University, Ottawa, Canada. 2007.