TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Aksel Salmistu 178082IABM

APPLYING STATECHARTS IN ENTERPRISE INFORMATION SYSTEM. A CASE STUDY

Master’s Thesis

Supervisors: Tõnu Näks
MSc
Margus Freudenthal
PhD

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aksel Salmistu 178082IABM

OLEKUSKEEMIDE KASUTAMINE ETTEVÕTTEINFOSÜSTEEMIS – JUHTUMIANALÜÜS

Magistritöö

Juhendajad: Tõnu Näks
MSc
Margus Freudenthal
PhD

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Aksel Salmistu

03.01.2020

Abstract

The goal of this thesis is to propose a new solution for describing processes automated by the Estonian Customs Information System (CIS). It is envisaged that the process defined for CIS will be useful for developing enterprise information systems in general. The current process for CIS development already takes advantage of using state machines, a custom-defined language for specifying these state machines and a applicable code generator. This thesis proposes to replace the currently used custom language with the statecharts formalism and select a more widely adopted tool for creating the specifications and generating code. In order to determine if the proposed solution has desired benefits, a case study is conducted on an actual customs information system. A suitable tool is selected based on the requirements of the system that is used to model the life-cycle of core objects of the customs information system. This tool is used to develop statechart models and generate code that is integrated to the rest of the system. Different aspects of such an implementation are discussed to show the benefits and provide solutions to some of the issues that were encountered during the implementation. The thesis will provide evaluation to the suitability of statechart models and answers if the issues regarding previous solution can be solved with the help of statecharts.

This thesis is written in English and is 40 pages long, including 5 chapters, 13 figures and 1 tables.

Annotatsioon

Olekuskeemide kasutamine ettevõtteinfosüsteemis – juhtumianalüüs

Lõputöö eesmärk on pakkuda välja uus lahendus Eesti tolliinfosüsteemi automatiseeritud protsesside kirjeldamiseks. Usutakse, et tolliinfosüsteemi arendamiseks mõeldud protsess on kasulik ka laiemalt, näiteks ettevõtte infosüsteemide arendamisel. Praegune tolliinfosüsteemi arendus kasutab olekumasinate kirjeldamiseks kohaldatud keelt ja vastavat koodi generaatorit. Käesolev magistritöö teeb ettepaneku asendada praegune kirjeldamise keel standardse Hareli olekuskeemi formalismiga. Selleks võetakse olekuskeemi spetsifikatsiooni koostamiseks ja koodi genereerimiseks kasutusele üks laialt levinud tööriistadest. Lõputöö käigus viiakse läbi juhtumianalüüs, et teha kindlaks, kas pakutud lahendusega kaasnevad soovitud tulemused. Selleks, et olekuskeeme kasutada on esmalt vaja valida sobilik tööriist, mis vastaks süsteemi nõuetele. Valitud tööriista kasutatakse olekuskeemide koostamiseks ja mudelite põhjal koodi genereerimiseks, misjärel integreeritakse genereeritud kood ülejäänud süsteemiga. Magistöös arutletakse sellise lähenemise erinevate aspektide üle, et näidata kasulikke vaatenurki ja pakkuda välja lahendused mõningatele probleemidele, millega pidi olekuskeemi kasutamisel silmitsi seisma. Seejärel hinnatakse olekuskeemi sobilikkust, et teada saada, kas eelneva lahenduse probleeme on võimalik nende abil lahendada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 40 leheküljel, 5 peatükki, 13 joonist, 1 tabelit.

List of abbreviations and terms

EIS	Enterprise Information System
MDD	Model-Driven Development
FSM	Finite-State Machine
BRMS	Business Rules Management System
CIS	Customs Information System
UI	User Interface
DSL	Domain-Specific Language
CASE	Computer-Aided Software Engineering
MASP	Multi-Annual Strategic Plan
UML	Unified Modeling Language
XML	Extensible Markup Language
IDE	Integrated Development Environment

Table of Contents

1 Introduction 11
1.1 EIS development 11
1.2 Problem identification 12
1.3 Customs information system 13
1.4 Model-Driven Development 15
1.5 Related work 18
1.6 Structure of this thesis 19
2 Statechart tools 21
2.1 Criteria 21
2.2 IBM Rational Rhapsody 22
2.3 MagicDraw 23
2.4 Enterprise Architect 24
2.5 Yakindu 25
2.6 Evaluation of statechart tools 27
3 Experience with the Yakindu tool 29
3.1 Current implementation of state machines in CIS 29
3.2 Example of statechart developed with Yakindu 32
3.2.1 Declaration statechart 32
3.2.2 The declarationTaxes statechart 37
3.3 Observations 41
3.3.1 Benefits of composite states 41
3.3.2 Troubles with the declarationTaxes statechart 42
3.3.3 Communication with client code 43
3.3.4 Storing and displaying the states of an object 43
3.3.5 Issues regarding user interface 44
3.3.6 Model simulation 45

3.3.7 Code generation . . . 46
4 Discussion . . . 47
5 Conclusion . . . 50
References . . . 51
Appendix 1 – Debtor statechart . . . 54
Appendix 2 – PersonTaxes statechart . . . 55
Appendix 3 – DebtorStatemachine.java class . . . 56
Appendix 4 – IDebtorStatemachine.java class . . . 74
Appendix 5 – IStatemachine.java class . . . 76
Appendix 6 – ITracingListener.java class . . . 78

List of Figures

Figure 1. Modeling perspective in Rational Rhapsody 23

Figure 2. Modeling perspective in MagicDraw 24

Figure 3. Modeling perspective in Enterprise Architect 25

Figure 4. Modeling perspective in Yakindu 26

Figure 5. DSL syntax 29

Figure 6. DSL syntax first example 29

Figure 7. DSL syntax second example 30

Figure 8. DSL syntax third example 30

Figure 9. Example state machine 30

Figure 10. Example state machine DSL 31

Figure 11. declaration statechart 33

Figure 12. declarationTaxes statechart 38

Figure 13. Simulation in Yakindu 45

List of Tables

Table 1. Statechart tool evaluation 28

1 Introduction

1.1 EIS development

Enterprise Information System is a software system which offers a service that supports organization's main processes and goals [1]. One of the main challenges of developing EIS is the complexity of automated processes that tends to grow fast as the analysis is progressing. While the EIS development is driven by humans, the decisions can be taken on the fly and business processes need to adapt to these decisions. Automating these processes means that all the possible situations have to be foreseen and the behavior formalized. Coupled with non-functional requirements on business critical systems such as scalability, flexibility, security, usability and adaptivity, the complexity may become overwhelming very quickly. Many of the EIS development projects result with a huge budget overrun, missed deadline or reputation damage [2].

Different agile development methods have gained popularity and are well justified when system can be decomposed to components easily. Unfortunately, in many software development projects like public procurement, the agile development process can not be used. In case of complex systems a full understanding provided by top-down analysis-based methods is still useful [3]. The first phase of top-down development is called the design phase and it usually starts with the collection of requirements from different stakeholders. These requirements must answer the question why is the system important and how can it satisfy the needs of the organization. System requirements must be documented to later validate if the expectations were met or not. This documentation can also be used to keep an eye on the scope of the project and help to solve potential claims. The next phase is the development of software during which the architecture is defined, features derived from requirements are implemented and integrated. The development phase can be split into multiple iterations and can be delivered to the client after each iteration. All the features have to be developed after which the phase of stabilization begins where the quality of software is evaluated. The final delivery to

the client is made and the software will be put to production. Although the software development process ends here, the need to provide high quality maintenance is also important.

1.2 Problem identification

Recently the National Audit Office of Estonia conducted an analysis on why the state's software development projects fail [4]. The report analyzed the development of nine state information systems. The main aim was to identify the general risks and use them to make recommendations on how to reduce or avoid problems in the future. Two relevant reasons why the projects failed emerged from this report:

1. Legislational changes – these changes must be made during the design phase, development phase or after the information system is completed

2. Maintenance was overlooked in the design phase – the design phase did not assess how to ensure the continuous operation of developed software and the support for institution's basic processes. In addition the cost for maintenance and support was not assessed

Based on these reasons the National Audit Office of Estonia provided three essential key factors for success:

1. The changes in legislation must be taken into account when legislation is prepared, so that the information system developments can be completed by set deadlines and it would be possible to implement the legislation

2. The processes of principal activities must be described and optimized before software developments, so that the software prepared can make the principle activities more efficient and offer reliable e-services

3. The people who will participate in the project must be trained before the software development project starts (e.g. explanation of what the software development process is, how it is carried out, who are the participants). The roles, tasks and responsibility of the participants in the projects must also be clarified in the initial stages of the project

When system needs to adapt to constant changes in legislation it means that system should be designed in a way the modifications in business logic can be taken into account with minimal effort during the development or maintenance phase. One of the ways the maintenance and support process can become tiresome is when the documentation of the developed system is not updated at the end of the project. Ideally, every time a change regarding the system's behavior is made or a requirement that influences the functionality is found, the document describing this part of the system should be updated as well. Unfortunately, this work is usually not done due to various reasons, mainly lack of time, money or regard. Situation where documentation and implementation do not correspond causes a lot of problems during the maintenance phase because it is difficult to determine how the system is currently working and how it should work. An up-to-date documentation is vital to make legislational changes that do not cause unwanted harm.

In addition to making the system adaptable for the maintenance, the last two previously listed key factors are important as well. The main processes of the system need to be described and analyzed before the development can begin and all stakeholders must share common understanding of these processes. This is usually difficult because the business and development side do not see things the same way and speak in different terms. The problem can be overcome with the help of understandable, clearly defined models that can serve as a basis for communication between the two groups. Ideally, every development team member should have a detailed understanding of the system behavior so that the impact of potential changes could be foreseen before the actual implementation. This objective is hard to reach without behavioral models because Java code can not be understood by every stakeholder. One of the systems where business logic constantly changes is customs information systems (CIS).

1.3 Customs information system

This section describes the overall functionality and specification of customs information systems. The nuances of customs information system are introduced to give readers a better understanding how they are used and why they are important. The general architecture of these systems is also described. Cybernetica AS has developed customs

information systems for Estonian Tax and Customs Board since 2003 [5]. The author of this thesis has participated in the development of these systems from 2016. The long history and competence in business domain will continue to grow in the future because customs systems must be upgraded and maintained to correspond to legislation.

Customs information systems are one of the subcategories of enterprise information systems. They can be described as systems that store and process information regarding customs operations. Over the years paper-based customs procedures have gradually decreased and electronic systems have taken over. In order to create European electronic customs environment a document named MASP (Multi-Annual Strategic Plan) was drawn up by European Commission and Member States [6]. MASP is used to justify budgetary requests made by national customs administrators and to give detailed overview of business and IT-technical aspects of new IT projects in the area of customs. The MASP covers development of numerous systems over a long period of time. Some of the systems described in MASP should be upgraded while others are completely new. These systems enhance competitiveness of European businesses and improve safety and security checks. Number of these systems are developed and maintained by European Commission while others are being developed, integrated and maintained by each Member State on their own. In Estonia a separate system for every electronic customs document has been developed, which holds all business logic related to that document. The general life-cycle of different documents is somewhat similar, so the overall architecture of these systems is similar as well.

One of the characteristics of customs information systems is that they are in constant change. This means that the business rules and the behavior of the system change frequently and therefore the system must be designed in a way where these changes can be carried out with minimal effort. Main reason why these systems are in constant change is legislational changes that cause information systems to change as well. Often these changes need to be carried out on a tight time frame because systems have to be ready when the legislation comes into force. As Estonia is part of the European Union and therefore a member of unified customs unit it means that legislation changes not only come from the national government but also from the European Commission.

From architectural perspective the customs systems are rather similar because they

handle electronic documents that are usually in XML form. Behind every document is a different process that usually starts with draft version and ends with final version where the processing of the document has ended. The different customs information systems communicate with each other via interfaces. The document life-cycle and the exchanged messages can be described with behavioral models such as state machines. In this case state transition can be triggered by a message from another system, a user (through user interface) or a timer (after some time passes). Although finite-state machines (FSM) are simple to understand and develop, they lack flexibility that is needed to describe a more complex system. Many of these state machines have to communicate with each other and a state change in one state machine can trigger a state change in a different state machines. The messages between different state machines are exchanged with the help of Java code. For smooth operation of state machines these messages must be implemented with caution to minimize the impact of potential mistakes. There is no easy way to test the exchange of these messages, it can only be made through the application's user interface. Customs information systems have a lot of business logic which is why the testing is not trivial. In order to trigger a state change the tester must understand business logic and find out which steps are needed to fire the trigger. In other words the testing of a business process in customs information system is very time-consuming and requires a good understanding of the general business requirements of the system.

A solution to these issues can be seen in model-driven development (MDD) whose main idea is that the central process and objects should be viewed and constructed as models. In that case the code describing this object/process is generated from these same models and this code is integrated to the system so that none of the details goes missing when the model is transformed to code and integrated to the system. It also means that the implementation and documentation are synchronized every time a change is made to the system.

1.4 Model-Driven Development

Models in software development process can be used for variety of purposes. One of the most common goal of a model is to allows stakeholders to communicate with each

other. An up-to-date model of a system's behavior can assure that software behaves as expected and meets the requirements.

Model-Driven Development is software development principle that is based on visual construction (models) of complex applications. MDD is often connected to the concept of platform-independent models because constructed models are not dependent on chosen computing technology and evolutionary changes. Constructing models are good way to understand and explain systems behavior but they have little value if used only for documentation purposes as documentation is rarely up-to-date with corresponding code. A key premises behind MDD is that applications can be automatically generated from their corresponding models [7]. MDD lets programmers to model program's functionality instead of spelling out every detail using a programming language. MDD aims to raise abstraction level so that software developers can specify what the program should do rather then how it should be done [8]. "Model-driven development enables reuse at the domain level, increases quality as models are successively improved, reduces costs by using an automated process, and increases software solutions' longevity" [9].

One of the models where MDD techniques can be applied is state machines. State machines can be used to describe the life-cycle of an important object or a process in the system. Unfortunately the classical state machine known as the finite-state machine (FSM) is not very scalable due to the phenomenon known as state explosion. This causes FSMs to become much more complex then the complexity of the system that it describes. Improved version of classical FSM is named statechart. Statechart or Harel's statechart is a model of states and transitions that is used to describe the behavior of the system. Statechart formalism addresses the problem of conventional FSMs so that the complexity of statechart no longer explodes in comparison to the system it describes. This means that state machines have truly applicable approach to real-life problems [10]. In addition statecharts have some extra features compared to FMS that makes them attractive way for describing behavior of complex systems like EIS. The most important features compared to FSM-s are:

- Composite states and regions – this allows states and regions to contain other states or regions. One composite state should be defined as the ultimate root of state machine hierarchy. Sub-states can share all aspects of behavior with their superstate

(for example sub-states can use the same transition to exit superstate). This feature of statechart allows it to cope with complexity as some of it is abstracted. Hierarchical states are a good way to make model more abstract because modeler can zoom in or out to view the model from different perspective (depending on how much detail is needed).

- History states – this allows to re-enter a composite state or a region at the point where it was left.

- Orthogonal states – this allows to run state machines concurrently. Orthogonal states allows a statechart to fragment into two or more independent parts that are concurrently active. Key benefit from orthogonal states is that they allow modeler to avoid mixing independent behaviors and keep them separate. Specification of statechart does not require each orthogonal region to be executed in separate thread. The specification requires that the modeler does not rely on any particular order in which an event will be sent to orthogonal state.

- Variables – this allows to memorize values and drastically reduce the necessary number of states.

- Actions – can not only occur along transitions, but additionally inside of states, especially as entry actions or exit actions.

- Activities – bridging the gap between a state machine and real-world behavior.

These features make statecharts an attractive way to describe the behavior of objects in a complex system. Using statecharts in the development of complex systems has many benefits. One of them is the ability to derive test cases from the model [11]. Once the model has been developed the testing process becomes less time consuming and more efficient. Generating code from statechart model minimizes the potential risk to make mistakes while implementing the behavior. Bugs can be found and fixed in less time, while using statechart to describe a process [12]. Further, the code generated from statechart can be re-used with very few adaptions based on the specific requirements. This can be especially useful for the development of mobile applications were different operating systems need to be supported [13].

Because statecharts have more complex construction than a classical FSM the implementation of the complex behavior can lead to mistakes being made while implementing the desired functionality. To overcome this difficulty, one should use a tool that can generate code from the statechart model. Such a tool can reduce the probability of bugs and avoid confusion from semantical differences in statechart version. This way, it is easier to share common view of the model and assert that the model corresponds to the implementation of the system. This thesis analyzes how can MDD, in particular statecharts, be used in customs information system development.

1.5 Related work

Statecharts have been used for variety of purposes for example to model the navigation process of a web page [14]. The paper proposes statecharts as a formalism to model different parts (hyperlinks) of a web page and describe the navigation process between pages. This work was done solely for modeling/documentation purposes and no code was generated from these models. Proposing a methodology for testing a statechart has also been a popular topic [15], [16]. These methods can be used to provide additional benefit for using statecharts, although the testing functionalities of various statechart development tools should also be considered because they can significantly simplify the testing process and increase the quality of the whole system.

Because of the practical outcomes of this thesis, publications where code is generated and implemented to the system are more comparable. One of the works with practical outcome is describing a behavior of a robot [17]. The authors developed a tool (named KSE) for modeling statecharts and to describe how robots should act in RoboCup soccer competition. In video game industry statecharts were used to to model the artificial intelligence behind a non-player character [18]. One of the most similar works to this thesis has been done by Sinha, Narula and Grundy [19]. In their work they use statecharts to implement the behavior of IoT apps. They introduce the concept of parametric statecharts to change the configuration of mobile app based on the number of sensors a smart home has. Generated code from statecharts is used to minimize the need to write same code for different apps. This thesis is aiming to change the behavior of

the system rather then the configuration. They are also focusing on developing IoT healthcare apps while current thesis focuses on complex enterprise information systems. Another practical statechart implementation was done in medical sector, where medical guidelines were transformed to statechart models in order to simplify the process of patient treatments [20], [21]. The statechart notation was basically used to model a decision tree in order to help medical workers to find a suitable treatment based on the symptoms patient is having. They saw that a statechart modeling tool could be used by a medical professional to change or develop medical best practice guidelines. Most of previous papers use statecharts formalism for completely different reasons.

The overview of previous research concludes that practical work focusing on statechart implementation has mainly been done in embedded software development. The MDD concept itself has been previously used for enterprise information system development. For example generated code from sequence diagrams was integrated to the English Auction protocol [22]. In addition a case study based on Statoil business processes was conducted to show how these models could be integrated to enterprise information system [23]. The author made effort to find similar works describing fully the business processes and using the models for code generation with statecharts, but a solution similar to that employed in CIS was not found. This thesis will serve as an extension to some of the papers previously published. It provides an example for using statecharts in complex document based systems and the proposed solutions can be used in other similar enterprise information systems.

1.6 Structure of this thesis

Chapter two introduces some of the most popular tools for constructing statecharts. These tools are evaluated against requirements of the customs information system to determine the most suitable tool for the given problem domain.

Chapter three shows how state machines have been previously developed in CIS and discusses the issues with this approach. Then the statechart formalism is used to describe a core object of CIS to show how the new process looks like. Chapter three also provides a detailed specifications of the statecharts developed during the case study. The author

will discuss the observations made during the integration process and propose solutions to some of the issues that had to be faced.

Chapter four discusses the negative and positive aspects of using statecharts in the development process and gives recommendations on what to consider when applying statecharts in other enterprise information systems.

Chapter five answers whether or not statecharts proved to be useful for the CIS development process.

2 Statechart tools

2.1 Criteria

This section describes the requirements for statechart development tool. These requirements must consider the general requirements of enterprise information systems and also take into account the nuances of customs information systems described in section 1.3. Based on the usual requirements of EIS and in particular customs information system, key requirements for the statechart development tools are determined:

1. Ability to visually construct a statechart – the tool should have a graphic interface for statechart modeling

2. Ability to validate and simulate statechart – the tool should provide a easy way to test the statechart without integrating the generated code to the system. This will help to test the behavior from the very beginning of development process

3. Ability to generate Java code from statecharts – the code generator must output standard Java code without using unnecessary external libraries

4. Cross-platform – the tool should be available for Windows, Linux and MacOS operating system

5. Support for complex statechart constructs – all key benefits of statechart formalism described in section 1.4 should be supported

6. Easy integration to the rest of the system – the integration process and nuances of the tool must be well documented

7. Open license or extended trial – the tool must be free of charge or provide an extended trial throughout the time needed to conduct the case study

There are a variety of tools that can be used to generated code based on a model like class-diagram, ER model or statechart. Because of this, the selection was narrowed

down to four most popular and advanced tools for statechart modeling. In the next sections, author will provide an overview of the four selected tools and later these tools are evaluated against the introduced requirements.

2.2 IBM Rational Rhapsody

One of the most popular and advanced tool for statechart modeling is IBM Rational Rhapsody [24]. IBM Rational Rhapsody is also one of the oldest tools for generating code from models with first version of software dating back to year 1996. Since then the product's ownership has changed hands multiple times, but the product has continued being refined and many features have been added. The Rational Rhapsody tool is for system engineers and software developers who want to create, document and test real-time or embedded systems or software. With the help of Rational Rhapsody development teams can: collaborate to analyze requirements, optimize design decisions and validate functionality early in the development life-cycle, perform design reviews, and automate the delivery of innovative, high-quality products. The statechart development ability is only a small fraction of the many features this tool has. The number of features is dependent on the edition being used. Figure 1 shows how the statechart modeling workstation looks like in Rational Rhapsody.

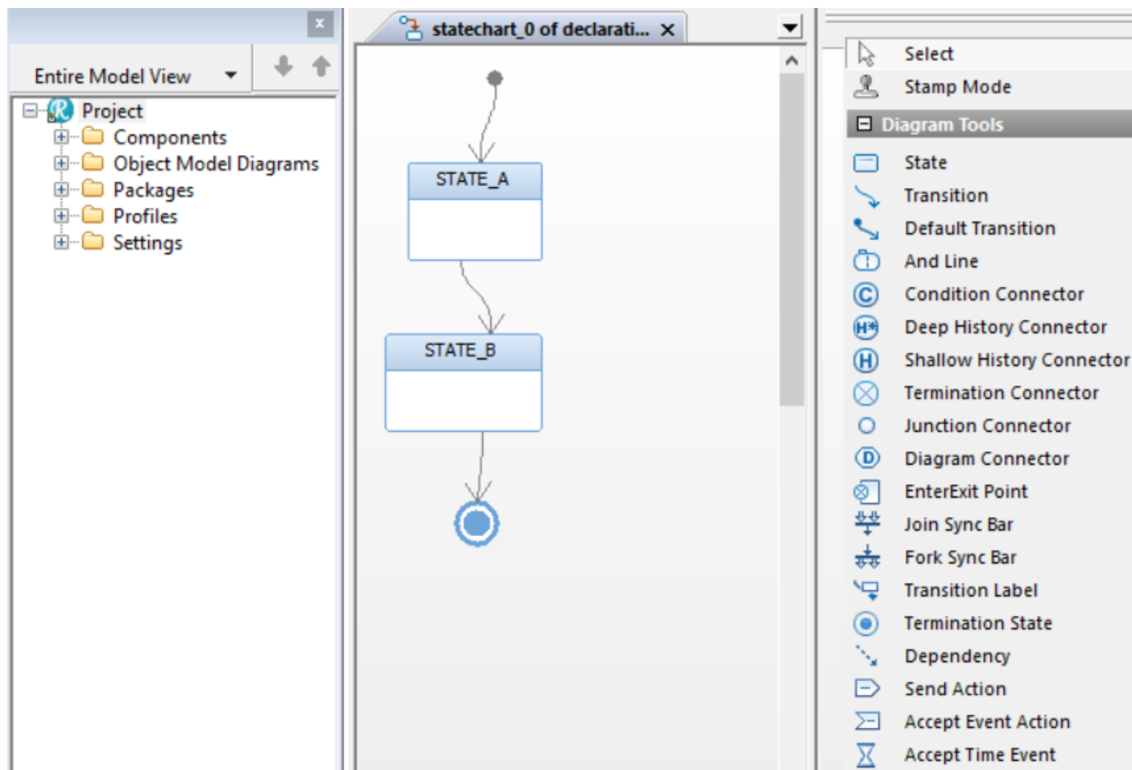

Figure 1. Modeling perspective in Rational Rhapsody

2.3 MagicDraw

Another powerful tool for statechart development is MagicDraw [25]. MagicDraw offers a lot of different features like source code reverse engineering, teamwork, making custom diagrams etc. It also fully supports UML 2.5 and possibility to make extensions like customizing the appearance of diagram elements. It can also be integrated with leading Java IDE's like Eclipse, IntelliJ IDEA, NetBeans and more. MagicDraw allow to generate a customized report from the model to ease the process of software design documentation. One of the features that separates MagicDraw from its rivals is that before the code artifacts are generated from model MagicDraw performs source code reverse engineering in order to synchronize changes in the source code with the changes in the UML 2.5 model. This allows to add comments to the model that are not tool specific. These comments have been the reason why many other tools fail to generate code from UML 2.5. Figure 2 shows how the statechart modeling workstation looks like in MagicDraw.

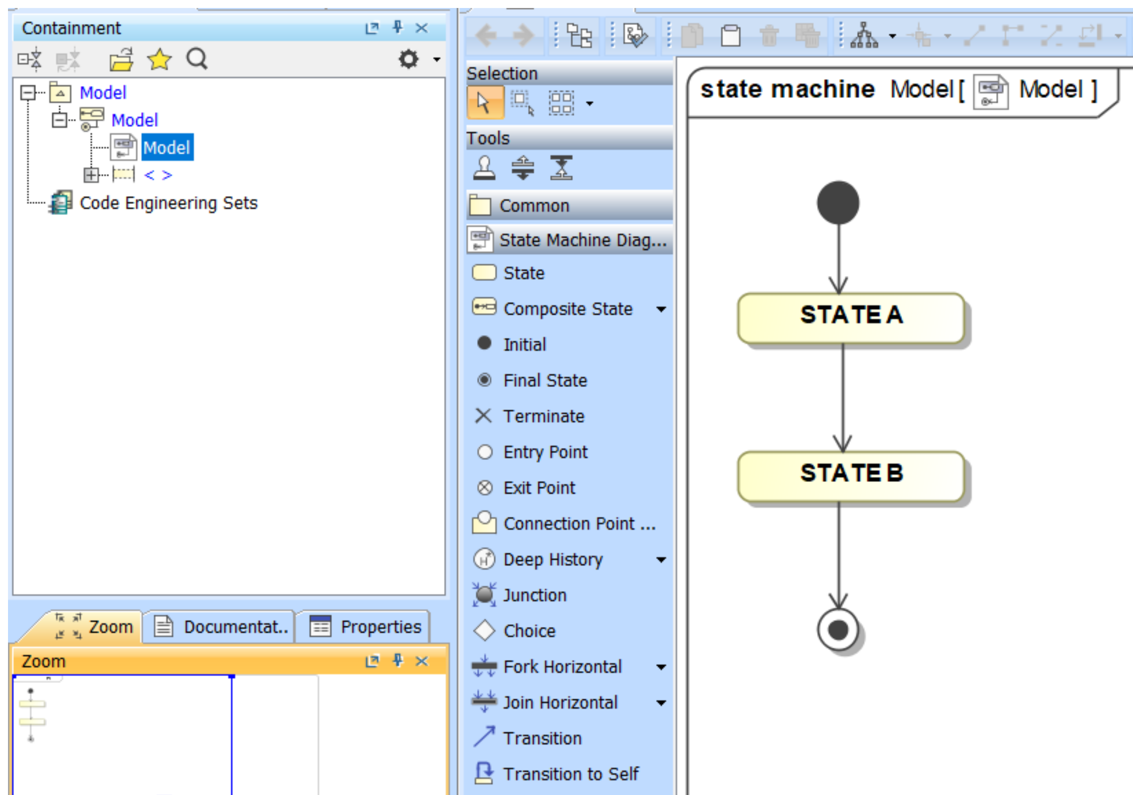

Figure 2. Modeling perspective in MagicDraw

2.4 Enterprise Architect

Enterprise Architect (EA) is one of the most popular CASE tools [26]. EA can be used to model and manage complex information by integrating and connecting a wide range of structural and behavioral information in visual form. This tool has two main features that separate it from rivals: baseline and version management for tracking and integrating changes, role-based security to help certain individuals contribute in specific way. In addition to modeling it can be used to generate code from following behavioral models: statecharts, sequence diagrams and activity diagrams. One of the drawbacks of the tool is that it lacks capability to run on Linux and MacOS systems, although it is possible while using Wine [27]. Figure 3 shows how the statechart modeling workstation looks like in Enterprise Architect.

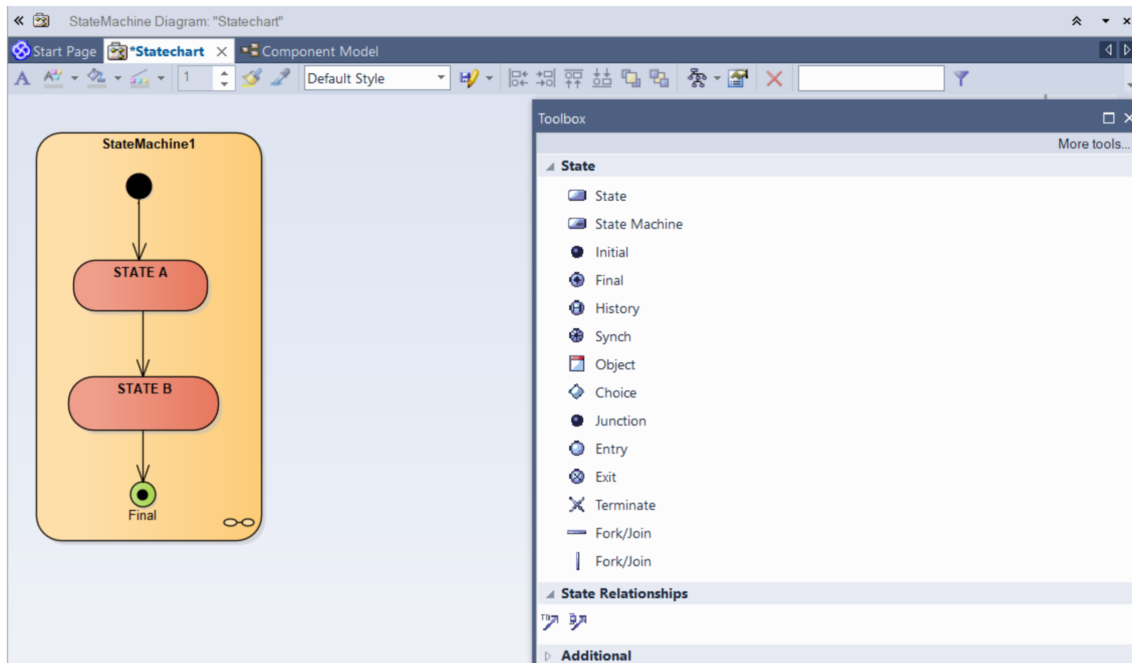

Figure 3. Modeling perspective in Enterprise Architect

2.5 Yakindu

This section describes statechart modeling tool Yakindu and shows how does it differ from the previously described tools. More in depth analysis is given on functionalities of Yakindu compared to classical Harel's statecharts. Yakindu is one of the most advanced tools for modeling statecharts [28]. Yakindu can be used as a stand-alone application or as an Eclipse plugin. The latter is open-source and free to use. Yakindu features graphical interface for modeling state diagrams. Other key features are validation of statecharts, simulation tool and code generator for state machines in Java, C and C++. A big advantage is that generated code does not rely on any additional runtime libraries.

A statechart model in Yakindu consists of two separate parts: canvas and definition section. Canvas is used to model the statechart by adding regions, states, transitions etc to it. Definition section is used to define objects used by the statechart, for example variables, events, operations etc. Canvas can host multiple regions. Purpose of a region is to group a statechart so that it becomes possible to have multiple state machines in different regions and run them simultaneously. Figure 4 illustrates the definition section, canvas and region concepts in Yakindu.

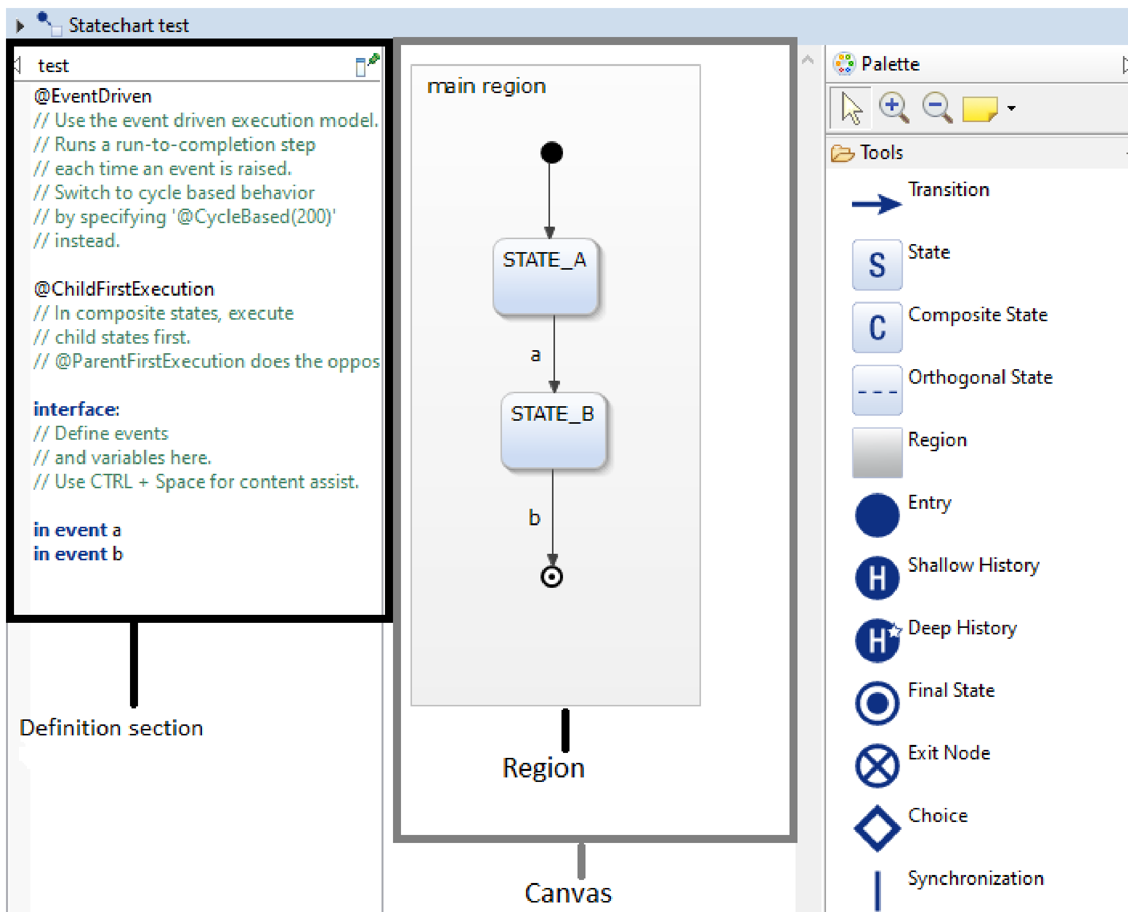

Figure 4. Modeling perspective in Yakindu

Behavior of a state can be used to pass arguments outside of statechart or to trigger a transition if condition becomes true. Transition is a transfer from source state to target state. Once a transition is taken, the source state becomes inactive and target state becomes active. Each transition should have a reaction because a transition without a reaction is never taken. Yakindu supports two types of statechart execution: parent-first and child-first. This can be specified by setting @ParentFirstExecution or @ChildFirstExecution annotation in the definition section. This allows developer to decide the checking order of states. In case of parent-first execution parent states are checked first to see if a transition can be taken. In child-first execution the sub-states are checked first. If no specification is given, Yakindu is using parent-first execution as default. Parent-first execution is the classical way of statechart prioritization proposed by David Harel, while the child-first execution is object-oriented way to handle priorities of transitions. Synchronizations enable to split or join a flow into multiple sub-states. A synchronization with one incoming transition and multiple outgoing transitions is called

forks and synchronization with multiple incoming transitions and one outgoing transition is called joins. Key benefit of using synchronization becomes evident when there are parallel flows or sub-states. It allows to execute statechart faster and to ensure that final state is reached only when all parallel flows have been completed.

Yakindu supports reactions syntax familiar from automata theory. Reaction syntax for all reactions is: **trigger [guard] / effect**. All components are optional but as stated before a reaction without a trigger has no effect. Triggers are used to activate a reaction. They can be defined as event or a predefined special trigger can be used (after, every, always, default etc). It is also possible to have multiple triggers on one reaction. To specify when reaction should be taken developers should use guard, which consists of a boolean expression that must be true in order to cause a reaction. Actions or reaction effects are events that occur after trigger is fired and guard condition becomes true. Reaction effect could be assigning a variable, calling a operation or raising an event. Effect can have multiple actions that have to be separated by semicolon. Conditional statements are supported in actions. Yakindu also offers a feature to simulate statechart model. In simulation the statechart is executed in order to find out if statechart acts as developer intended or not. During a simulation developer can interact with a simulation by passing/modifying values of variables or by manually raising events. Yakindu Professional Edition simulation also features possibility to add a breakpoint to states or transitions and to create snapshots of the simulation.

2.6 Evaluation of statechart tools

This section evaluates the tools described in the previous sections against the requirements described in section 2.1. The author will select the most suitable tool for constructing statecharts that will be used in the development process.

All tools evaluated have the possibility to visually construct a statechart and validate the designed behavior through validation. All tools also provide code generation for different languages including Java. Only one tool did not fully meet the cross-platform requirement. Considering that many developers in Cybernetica use Linux or MacOs systems and the limitation of Wine usage the Enterprise Architect software was ruled

out. In addition the statechart development documentation of EA is very superficial and consists of only a few examples. MagicDraw and Rational Rhapsody are very similar with each other: they meet most of the requirements proposed in section 2.1 and they both can be used to construct different models and generate Java code from them. From this perspective Yakindu is a bit different because it is developed to be only used for statechart modeling. This difference can be useful because learning to properly use such a powerful tool like Rational Rhapsody or MagicDraw can be time consuming. MagicDraw and Rational Rhapsody both offer a trial version of the software but considering that it takes more then 30 days to validate whether or not statecharts can be used in the development process, a license of the software should be bought. Considering the cost of the software Yakindu stands out as the best choice from the tools evaluated because Eclipse plugin version of the software can be used for free. Purchasing a license should be considered if the desired benefits are met or code artifacts of other models are seen as a useful resource in the development process. Because MagicDraw and Rational Rhapsody do not offer an extended trial period, the Yakindu Statechart Tools was declared as the best tool for the problem domain and it was used in the development process. Table 1 illustrates the result of the evaluation.

	Rational Rhapsody	MagicDraw	Enterprise Architect	Yakindu
Graphic interface	yes	yes	yes	yes
Validation and simulation	yes	yes	yes	yes
Java code generation	yes	yes	yes	yes
Cross-platform	yes	yes	no	yes
Complex statechart constructs	yes	yes	no	yes
Easy integration	yes	yes	yes	yes
Open license/extended trial	no	no	no	yes

Table 1. Statechart tool evaluation

3 Experience with the Yakindu tool

3.1 Current implementation of state machines in CIS

This section describes the previous solution that was used to develop state machines. Author is describing previous experience from the systems analysts perspective and points out negative aspects. Author will also provide suggestions on how this solution could be improved.

In the previous solution the state machine development process started with the design of state machine model using CASE tool. After refining the model, state machine logic was written in an in-house Domain Specific Language (DSL). The specification written in the DSL was stored as a plain text file and converted to Java code during the build process. Figure 5 describes the main syntax of the DSL to give a better understanding on how the specification should be written.

```
(set-state STATE1) = sets state machine to desired state (STATE1)
(has-state condition*) = raises error when condition is not true
(state?  condition*) = triggers operations when condition is true
  has-state and state? describing the conditions of (states)
    (any STATE1 STATE2 ....) =  true when the state machine is in one
                 of the states in the list
    (not STATE)  = true when state machine is not in the named state
    STATE = true when state machine is in the named state
```

Figure 5. DSL syntax

Figures 6 and 7 show how events can be defined and transitions invoked using DSL.

```
(event someEvent()
        (state? (any STATE1 STATE2)
                (set-state STATE3))
        (state? STATE4
                (set-state STATE5)) )
```

Figure 6. DSL syntax first example

```
(event someEvent()
        (has-state (any STATE1 STATE2)
                (set-state STATE3)) )
```

Figure 7. DSL syntax second example

It is also possible to define operations that need to be taken when a state is reached, as illustrated in figure 8.

```
(on-enter STATE1
        (someOperation))
```

Figure 8. DSL syntax third example

Example below (figure 9) gives better understanding of DSL and how to model the design and write corresponding implementation in DSL.

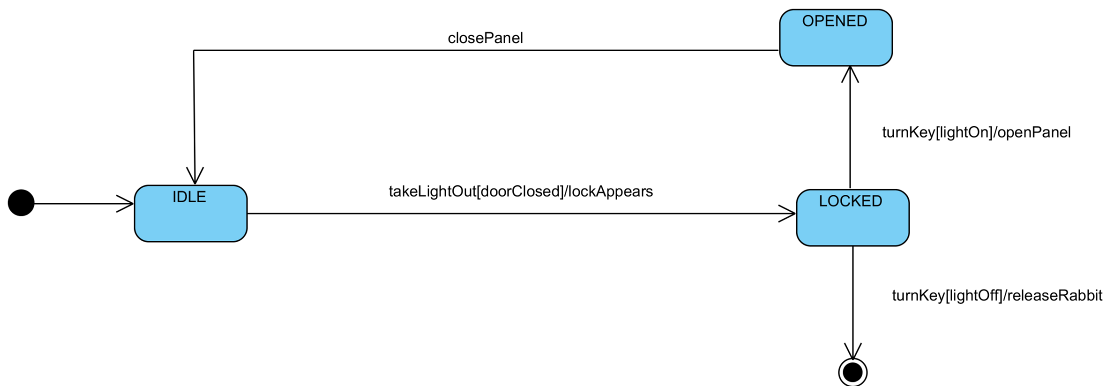

Figure 9. Example state machine

Based on this model the specification in DSL should be written like specified in figure 10.

```
(classificator secretPanelState
        # classifier description
        (desc 'Classifier for secretPanel states')
        (values
                (NO_STATE 'Initial pseudostate')
                (IDLE 'Waiting')
                (OPENED 'Opened')
                (LOCKED 'Lock closed')
                (END 'Final state')))

(fields
        (classificator state panelState (system))
        # when necessary more fields can be added )
(state-field state)
(event takeLightOut()
        (has-state IDLE)
        (if (doorClosed)
                (set-state LOCKED)))
(event trunKey()
        (has-state LOCKED)
        (if (lightOn)
                (openPanel)
                (set-state OPENED)
        (else
                (releaseRabbit)
                (set-state END)))
(event closePanel()
        (has-state OPENED)
        (set-state IDLE)
```

Figure 10. Example state machine DSL

The main downside with this approach was that the models were rarely updated after changing DSL, which caused occasionally issues as users discovered errors and developers had to figure out the proper behavior. The DSL file itself is rather easy to understand and issue is easily identified when searching for the problematic state or event. Understanding the complete life-cycle of an object becomes problematic once the model is more complex. In addition the DSL does not support more complex constructs of statechart formalism that could be useful for describing more complex process. Another downside becomes evident when the system maintenance is passed to another company who does not know anything about the DSL that has been developed in-house. To make harmless changes to system, new company must first learn and understand the DSL. The model written in DSL can not be simulated or validated for semantical errors. This means that in order to test if the state machine works as expected it must be integrated to the rest of the system. The issues with current solution can be summarized as following:

- State machine documentation does not correspond to the implementation
- State machines are difficult to read and develop without a graphical interface
- Using custom language for state machine development can cause vendor lock-in
- The current state-machine language is not sufficiently scalable

3.2 Example of statechart developed with Yakindu

This section provides an example on statecharts that were developed using statechart formalism and integrated to the customs information system. Author introduces two of the most important objects of a customs information system and their life-cycle is depicted as a statechart model. This section also introduces technical specification of these objects and describes the process based on the developed statechart.

Some of the state machines previously separately designed could be put together using composite and orthogonal states. On the other hand some statecharts need to be kept independent because they do not have a distinct relationship with each other or one object can have multiple number of these sub-objects. Only two statecharts developed while writing this thesis will be introduced because they are sufficient to demonstrate the solution and others are not that significant. Additional two statecharts developed can be seen in appendix 1 and appendix 2.

3.2.1 Declaration statechart

The most important and one of the most complex statecharts of CIS is the declaration statechart. Declaration statechart describes the whole process of a customs declaration from first draft version to the end where goods declared can be released for free circulation. The logic behind process and composite state marked with "X" will not be discussed. Figure 11 illustrates the declaration statechart designed in Yakindu.

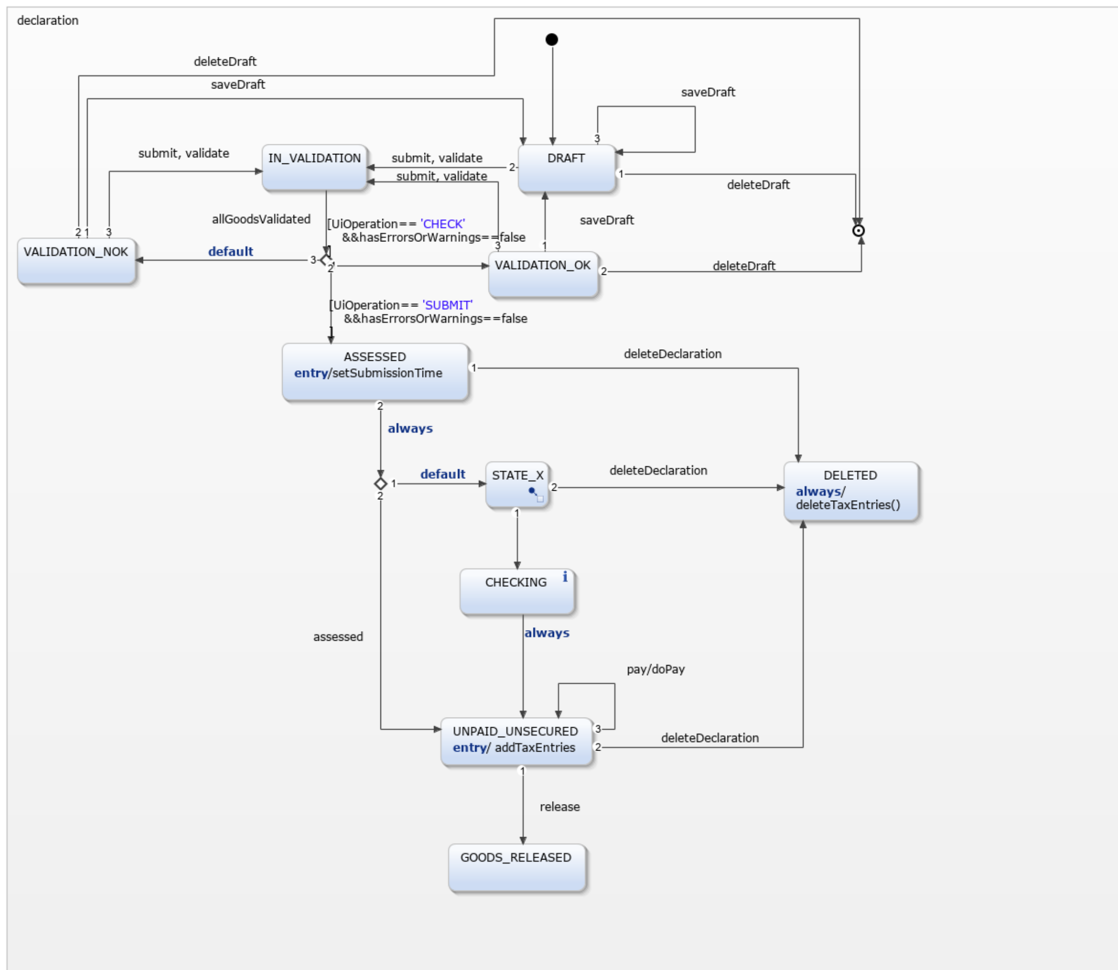

Figure 11. declaration statechart

To create a new declaration user must enter the user interface of CIS and press the button for creation, which will construct the declaration object and initiate statechart. The declaration statechart has following states:

- **DRAFT** – reached when declaration object is created, it is also the initial state. In this state user provides all data required to submit the declaration. This state is exited when:
  - draft is deleted - *deleteDraft*
  - draft is presented for submission - *submit*
  - draft is presented for validation - *validate*
- **IN_VALIDATION** – reached when user has provided all necessary data to submit or validate the declaration and system runs the business rules. Business rules

have been constructed using Business Rules Management System (BRMS) named Drools [29]. In order to indicate the desired behavior to the statechart, business rules must assign different variables. The main aim of the validation process is to detect errors or warnings regarding declaration data. This state is exited when:

  - the BRMS validation process ends - *allGoodsValidated*

- **VALIDATION_OK** – reached after the business rules have not detected any errors and if user just wants to validate the declaration but not yet to submit it. This state is exited when:
  - draft is deleted - *deleteDraft*
  - draft is presented for submission - *submit*
  - draft is presented for validation - *validate*
- **VALIDATION_NOK** – reached after the business rules have detected errors or warnings among declaration data. This means that user has to make modifications to the declaration before it can be submitted. This state is exited when:
  - draft is deleted - *deleteDraft*
  - draft is presented for submission - *submit*
  - draft is presented for validation - *validate*
- **ASSESSED** – reached when declaration is submitted. This state can not be reached before Drools validation confirms zero errors concerning the declaration object. This state is exited when:
  - declaration is deleted - deleteDeclaration
  - process x is needed - default transition
  - process x and tax entries are not needed - *assessed*
- **STATE_X** – reached after declaration is assessed. This is a composite state that holds all relevant information of process X

- **DELETED** – reached when declaration is deleted. After **ASSESSED** state is reached delete operation does not go to the final state because *declarationDeleted* operation does not actually delete all record associated with the declaration object from the database while *deleteDraft* actually does it

- **CHECKING** – reached after STATE_X. In this state customs officials define and execute control tasks. These tasks are assigned to one or multiple customs offices. In future development this state will be composite state like **STATE_X**

- **UNPAID_UNSECURED** – reached when system finds tax liabilities of declared goods during validation process. This means that tax entries based on the total amount of taxes should be added to an external accounting system. Entry to this state triggers an independent statechart declarationTaxes for which detailed explanations can be found in subsection 3.2.2

- **GOODS_RELEASED** – reached when goods declared have been released to free circulation and the declaration life-cycle has reached its desired state

Following is a more detailed description of reactions and variables used in declaration statechart:

- *validate* – event from UI indicating that user wants to present the declaration object for validation

- *submit* – event from UI indicating that user wants to present the declaration object for submission. Main difference between *validate* and *submit* is that in case of *submit* the declaration's object can no longer be changed by the user (provided that no errors were discovered during validation)

- *allGoodsValidated* – event triggered by the BRMS to signal the end of validation process

  - When *uiOperation* = "CHECK" and *hasErrorsOrWarning* = FALSE the statechart goes to state **VALIDATION_OK**

  - When *uiOperation* = "SUBMIT" and *hasErrorsOrWarning* = FALSE the statechart goes to state **ASSESSED**

  - In other cases the statechart goes to state **VALIDATION_NOK**

- *saveDraft* – event is triggered by the UI every time user changes data of the declaration before submitting. Save is triggered by various reasons for example exiting from a form field, session expiration, closing browser or pressing the save button. Statechart does not need the info on why declaration was saved rather then when
- *deleteDraft* – event is triggered by UI when user wants to delete the draft. All data regarding the draft is deleted from the database
- *deleteDeclaration* – event is triggered by the UI when user wants to delete the declaration. No data will actually be deleted and the declaration object goes to its final state **DELETED**
- *assessed* – event that can be used used when the declaration object bypasses **STATE_X** and checking activities. This can be used when a supplementary declaration is presented
- *paid* – event is triggered by the UI when taxes were not automatically paid and user wants to retry the payment process. This can happen when there is not enough money on user's prepaid account. After the money transfer user needs to indicate to CIS that payment process should be retried
- *release* – event is triggered by declarationTaxes statechart to indicate that taxes have been paid and declaration can be released
- variable *uiOperation* – shows which operation has been triggered in previous transition (*validate* or *submit*)
- boolean *hasErrorsOrWarnings* – is set to true when BRMS has found an error or warning regarding declaration data
- *setSubmissionTime* – operation is used to save the acceptance date of the declaration object
- *addTaxEntries* – operation is used to initiate the declarationTaxes statechart and start the payment process

- *deleteTaxEntries* – operation is used to delete tax entries made to external accounting system. Tax entries are deleted when they have been constructed but not yet sent to the accounting system. In case tax entries have already been sent new entries with negative values are made in a way that the overall balance will be zero.

3.2.2 The declarationTaxes statechart

The declarationTaxes statechart holds all logic regarding the interaction between CIS and external accounting system. The main goal of this statechart is to make entries to accounting system and verify that all taxes that occurred during the business rule validation have been paid for. This statechart is triggered by the declaration statechart when it reaches state **UNPAID_UNSECURED** and makes operation *addTaxEntries*. After taxes have been paid, system sends a message to declaration statechart and declaration object moves to it's final state in which all goods declared can be released and statechart goes to state **GOODS_RELEASED**. Figure 12 illustrates the declaration statechart designed in Yakindu.

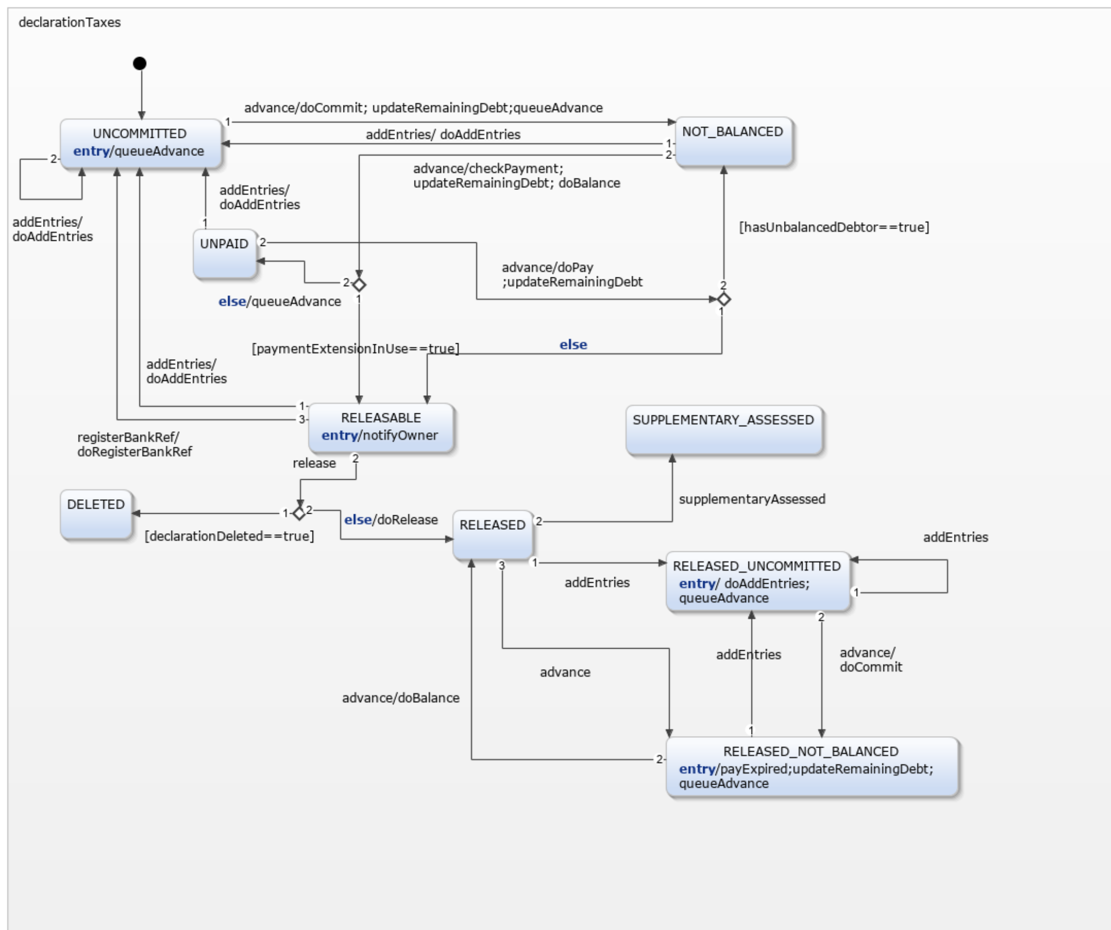

Figure 12. declarationTaxes statechart

System constructs the object and initiates declarationaTaxes statechart when declaration statechart raises operation *addTaxEntries*. The declaration statechart has following states:

- **UNCOMMITTED** – tax entries to the external accounting system are being made. This state is exited when:
  - all data regarding taxes is sent to the accounting system - *advance*
- **NOT_BALANCED** – balancing of guarantee amount is being done. When the amount of guarantee is less then needed the additional sum is guaranteed. The state is exited when:
  - guarantee amounts are in balance – *advance*
  - modifications to declaration have been made – *addEntries*

- **UNPAID** – taxes are being paid. When payment extension is not used user should pay taxes before the goods can be released. This state is exited when:
  - all taxes have been paid – *advance*
  - modifications to declaration have been made – *addEntries*
- **RELEASABLE** – the declaration object is releasable. This state is exited when:
  - declaration can be released – *release*
  - modifications to declaration have been made – *addEntries*
  - bank reference must be generated – *registerBankRef*
- **RELEASED** – declaration object has been released. This state is exited when:
  - supplementary declaration has been submitted – *supplementaryAssessed*
  - post entry or cancellation request has been submitted – *addEntries*
  - payment extension was used and the payment date was reached – *advance*
- **DELETED** – declarationTaxes object reaches this state when declaration is deleted, it is also a final state of declarationTaxes statechart. Transition to this state is decided after *release* event. It is also a final state of declarationTaxes object
- **SUPPLEMENTARY_ASSESSED** – a supplementary declaration has been submitted. It is also a final state of declarationTaxes object
- **RELEASED_UNCOMMITTED** – post-entry or cancellation request has been made to the declaration. This state is exited when:
  - all new tax entries to the accounting system have been made – *advance*
- **RELEASED_NOT_BALANCED** – guarantee amount is being balanced. This state is exited when:
  - all new guarantee amounts are balanced – *advance*
  - post entry or cancellation request has been submitted – *addEntries*

Reactions and variables of declarationTaxes statechart:

- *addEntries* – this event is used to change declarationTaxes state and trigger *doAddEntries* callback
- *advance* – this event is used to move the declarationTaxes object to more stable state. When no transition can be made this event is ignored
- *release* – this event is used to indicate that all tax entries have been registered in the accounting system and declaration can be released
- *supplementaryAssessed* – this event is used by declaration statechart to indicate that supplementary declaration has been submitted
- *registerBankRef* – this event is used to indicate that bank reference number should be provided
- boolean *hasUnbalancedDebtor* – variable is set to true when the debtor sub-machine (described in appendix 1) that is associated with declarationTaxes statechart is in state **NOT_BALANCED** or **RELEASED_NOT_BALANCED**
- boolean *declarationDeleted* – variable is set to true when declaration is deleted prior to the time when it could be released
- boolean *paymentExtensionInUse* – variable is set to true when user has a permit that allows to release declaration without previously paying all taxes
- *doCommit* – operation is used to send an event to sub-machines whose end result should be making tax entries to the accounting system
- *notifyOwner* – operation is used to send a release message to all sub-machines indicating that declaration can be released
- *queueAdvance* – operation is used to add event *advance* to message queue
- *doAddEntries* – operation is used to add tax entries to the external accounting system every time a new version of the declaration has been made and as a result tax differences are detected. When making modifications or post-entry, user will be submitting tax differences. Submitting a cancellation request will create negative tax entries based on the last declaration version

- *doRegisterBankRef* – operation is used to set the bank reference number. When bank reference number is not provided it will be generated
- *doRelease* – operation is used to set the payment date of the tax entries and move personTaxes sub-machine (described in appendix 2) to state **PAID**
- *updateRemainingDebt* – operation is used to calculate the remaining amount that must be paid based on the tax entries
- *checkPayment* – operation is used to send *declarationPaid* event to debtor sub-machine when the debtor(s) of the declaration do not have previous debt
- *doBalance* – operation is used to send balance message to sub-machines
- *doPay* – operation is used for paying for the declaration. This operation will not succeed when there is not enough money in the prepayment account
- *payExpired* – operation is used to call procedure from the accounting system that attempts to pay all expired tax entries for a given declaration

3.3 Observations

This section describes issues and aspects that need to be faced while developing the introduced statecharts. Author will discuss the obstacles faced and knowledge gained while implementing statecharts to the rest of the system.

3.3.1 Benefits of composite states

In the previous solution every state machine was designed as independent and to be exchanging messages with other state machines Java code had to be used. The relationships between the separate state machines were not explicit because they were described in different DSL files. This issue was overcome with composite states in Yakindu. Composite states allow to hide some complexity of the statechart model and only show the currently interesting part. For example process X was designed using a composite state to provide a better understanding how this process fits to the overall

life-cycle of declaration object. Not all independent state machines can be constructed as composite states because they have no distinct relations with the declaration object or declaration object can have multitude of these sub-objects.

3.3.2 Troubles with the declarationTaxes statechart

In the first version of developed statecharts the declarationTaxes statechart was a part of declaration statechart because logically they belong together and a declaration object can have ultimately one declarationTaxes object. Due to the overall architecture of CIS and nuances of the accounting system this approach was not very sustainable because all logic behind the accounting system was put into a separate service module. Almost all modules of CIS are dependent on the statechart module but in order to communicate with accounting system the statechart module itself should be dependent on the service module. It made more sense to place the declarationTaxes statechart in the same module as the communication with accounting system in order to make methods regarding accounting system only available to the declarationTaxes statechart. Fortunately, the communication between the two statecharts consists of few events and reactions. The state values of the declarationTaxes statechart were also saved to a different database table. Extracting one statechart from a bigger statechart model is made very convenient in Yakindu. The whole process consists of creating a new statechart and moving the desired states, transitions and events from one model to the other. It is also necessary to modify code generation project to add the new statechart and specify where the code artifacts should be generated. The declarationTaxes statechart also uses variety of callbacks that in most cases match corresponding events. This irregularity (compared to the rest of the statecharts) was again done due to the external accounting system. Because the accounting system is such a complex legacy system and preferably any additional complexity should not be added, the interface needed to remain unchanged. These callbacks were used in order to provide consistency between data stored to accounting system and statechart state. In previous solution the DSL allowed to use same name for event and operation. Because this is not supported in Yakindu and can cause confusion the *do* prefix was added to callbacks of declarationTaxes statechart.

3.3.3 Communication with client code

In order to integrate statecharts to the rest of the system, generated code should communicate with the client code. Yakindu offers three different options for a statechart to communicate with the client code: *in event*, *out event* and *operation*. *In events* are raised outside of the statechart implementation to indicate that a transition should be made. An *out event* sets a boolean flag to true when event is raised within a statechart, the client code must check the value of the flag to become aware that it needs to do something. An *operation* creates a callback instead of setting a boolean flag. From the modeling perspective the difference between an *out event* and *operation* are not very substantial. An *out event* must be raised with a raise keyword while *operation* is raised like a normal action (during a transition or within a state). In addition an *operation* can have multiple parameters while events can ultimately have one. In development process *in events* were widely used by client code to trigger a transition of the statechart. *Out events* were not used in the implementation although according to official documentation they should be used to communicate with the outside world. Instead of this *operation* was seen as a more suitable candidate for statechart to indicate that process needs to be initiated outside of statechart domain.

3.3.4 Storing and displaying the states of an object

State values of an object are useful resource for users because it can indicate that some processes are underway. In Yakindu a state is a vector and these states need to be translated to a readable form that could be displayed to the users. In addition, different states can be shown depending on the role of the user (client or customs official).

As a solution, first a database table named *status_change* was designed to hold all data regarding statechart execution. This table includes the current state, version of the state, timestamp and variable values of the statechart. Many of these states are purely technical and have little value to users but they can serve as a log that can be useful for debugging the system. Yakindu's tracing functionality saves the state value after making a transition. Enabling this functionality allows code generator to generate callbacks for every time a state is entered or exited. This callback allows to store the states of a statechart into

database. The vector of a state means state enum consists of all the region names and states that were passed through in order to reach the desired state, the pattern for state enum is: *region1_substate1_region2_substate2* and so on. For example, the state enum of the declaration statechart can be *declaration_ASSESSED*. For composite states this vector can become very long and difficult to understand. To overcome the issue with these obscures states an internal classifier was created to map the state vector to the state that needs to be displayed to the user. This allowed to hide the region and parent state prefix of a composite state from the user. This solution was also used to filter out state vectors that are used only within the statechart and are of a little interest to users. Due to regulations some states can not be shown to clients but need to be displayed to customs officer so a corresponding attribute was added to this classifier. Implementing such an internal classifier allowed to address all these issues.

3.3.5 Issues regarding user interface

Another related issue concerns the user interface of the CIS. When a user views declarations in CIS, many different buttons need to be displayed. The number of buttons is dependent on the current state of declaration's statechart. For a better user experience there is no need to show buttons that have no effect. These buttons are basically transitions that can not be taken from the current state. To achieve this functionality a method for running a dummy statechart was constructed in order to know which transition could be taken from the current state. Every time user interface needs to know whether or not to display a button, it would run the dummy statechart to know if a transition can be taken. These transitions were mapped to buttons and access permissions of the UI. This allowed to show users only buttons that have real impact on the declaration object in current state. In addition UI needed to be informed when state change had occurred that was not triggered from user interface because users want to know the latest state of declaration. UI was constructed to poll the current state of declaration based on latest state and refresh the declarations table if a change was detected. This allowed to show users up-to-date information on the declarations that were viewed from the table. In the old solution the users had to manually refresh the page in order to become aware of current status of the declaration.

3.3.6 Model simulation

While implementing statecharts one of the features that proved to be very useful was Yakindu's built in tool to simulate the statechart. It is the easiest way to verify if constructed statechart works as expected or not. The simulation process is rather easy – user has to manually fire triggers and assign variables in order to trigger a state change. Prior code generation is not needed to simulate the statechart. The active state is depicted as yellow and every time trigger of a possible transition is fired, corresponding state becomes active. If statechart is modeled as *CycleBased* the statechart cycles through to see if some condition has changed and makes transition when needed. In development *EventDriven* execution was used which means that in order to trigger a state change in simulation the event must be raised manually. The simulation allows developers to easily test if statechart acts as intended rather then integrate it with client code and then test the model through user interface. Figure 13 illustrates the simulation perspective of Yakindu.

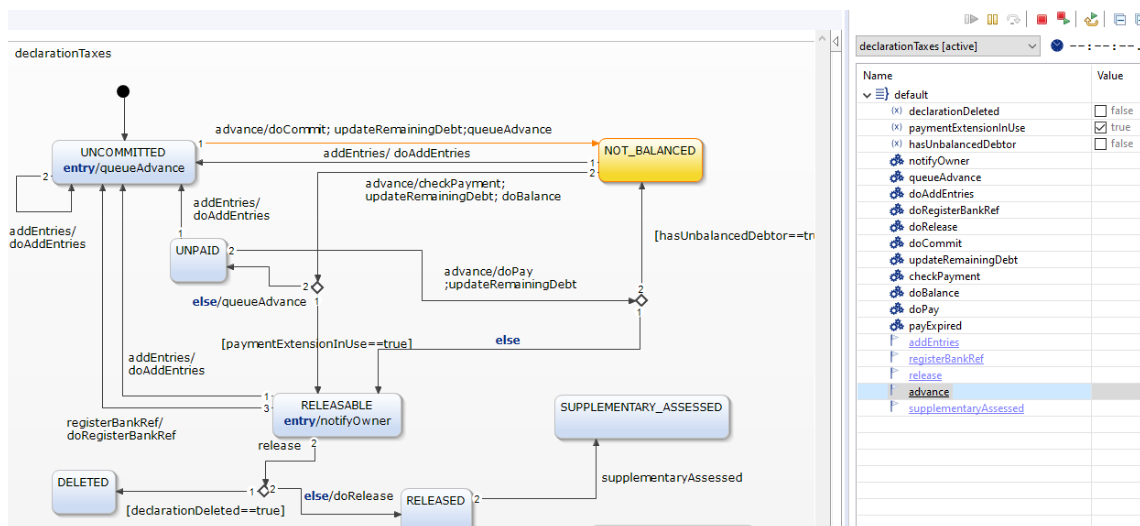

Figure 13. Simulation in Yakindu

In addition Yakindu actively validates the semantics of modeled statechart and shows error messages. Simulation can be run with some errors in order to simplify debugging but code can not be generated if errors exists in the model. This validation looks for semantic errors and also for variables, operations and events that are used in the model but not defined in definition section.

3.3.7 Code generation

The code generation process in Yakindu is made very easy. The user needs to create a code generation file based on the constructed model(s). Code generation model allows users to specify how and where should the code be generated. One generator file can be used to generate multiple independent statechart models. The generation model has many features that allow to specify different aspects of generated code. Most useful features for the development were:

- Naming – allows to specify the package name of generated Java classes
- Tracing – if enabled, generates the tracking callback functions (function for entering and exiting state)
- Outlet – allows to specify the target project and folder for the generated artifacts

When the project in Eclipse is set to automatically build, IDE generates code artifacts every time user saves the model. This feature rules out possibility to change the model without generating the code artifacts. The amount of generated artifacts is dependent on the overall complexity of the statechart. For example developed statecharts did not have any timed events and therefore the Java classes responsible for it were not generated. Based on debtor statechart the generator generated four different classes:

- DebtorStatemachine – holds all logic behind the statechart (state definitions, transitions etc.). Code example is provided in appendix 3
- IDebtorStatemachine – consists of all interfaces and events/callbacks that were defined under that interface. Code example is provided in appendix 4
- IStatemachine – holds the basic interface of the statechart, for example *init()* that initializes the state machine. Code example is provided in appendix 5
- ITracingListener – holds tracing interface of the statechart. Only generated when tracing feature is enabled. Code example is provided in appendix 6

First two classes are generated to every independent statechart model but both latter ones are used by every generated statechart (for example declaration and declarationTaxes).

4 Discussion

Overall it can be concluded that statecharts provide a good solution for describing the system's behavior in a complex system. A statechart model proved to be a good communication basis between developers and stakeholders of the business side. The benefits became more evident inside the development team when code was integrated to the rest of the system which sparked discussions over the whole process. Clear benefit of using statechart is the synchronization of the documentation and statechart model. This aspect will have positive implications during the maintenance period. The maintenance will also benefit from the fact that modifications to the statechart model can be done easily. All reactions and variables previously defined can be used to change system's behavior (for example adding a state) without the need to add additional Java code.

A potential drawback of statechart implementation is that all stakeholders must learn to understand them properly. In this aspect classic state machines are easier to understand and develop because they do not have complex constructs such as history, orthogonal and composite states. Team held many meetings to discuss the technical nuances and what is the best option to handle them. Now that these nuances have been addressed the implementation of statecharts of other similar document-based customs information systems will take significantly less time. On the other hand some of these complex constructs do not have to be used at all but they can be helpful to address some issues while describing the process. Many of the observations made in section 3.3 can be used in various systems but the technical requirements depending on the system must be considered before the same solution to these issues can be applied.

During the development process Yakindu proved to be a good tool for statechart modeling. No big issues were experienced except occasional error message when switching between git branches, which was probably caused by a bug in Eclipse plugin. Yakindu also comes as a standalone application that could potentially be more refined and stable but it can not be used without a valid license. As statecharts proved to be

useful for the development of CIS, the purchase of such a license should be assessed. Other tools besides Yakindu should be considered as well because as the evaluation of tools in section 2.6 showed they support a wider range of different models. One should note that semantics between various statecharts formalisms are somewhat different when deciding on using a different tool [30]. In order to use developed statecharts in other development tool the semantical differences require analyzing to evaluate precisely which aspects differ. The main risk is that although the semantics seem identical and the code compiles, the real behavior of the system changes. This risk can be minimized with thorough documentation of the new tool and ability to validate/test the statechart so behavior could be compared to Yakindu's simulator.

Compared to existing DSL the statechart model is simpler to understand, provides more flexibility and the life-cycle in general can be understood more easily. The developed statecharts are editable through graphic interface instead of writing implementation into a file. This means that it takes less time to learn developing statecharts when this task is assigned to someone with less experience. The learning curve also reduces because the statechart model can be easily validated for semantical and conceptual errors. Because Yakindu is well documented and available to anyone interested, the maintenance of the system can be passed to another competent company. The generated artifacts are standardized Java code and for this reason the needed changes to the system could be made directly in Java when for some reason Yakindu could not be used. Using statecharts instead of DSL proved also useful when more complex constructs of the formalism were used. Implementing such constructs in DSL would require significant development effort and could result in different interpretation of the semantics.

The main concepts, methods and processes for using statecharts in the development of CIS were developed and taken into use. Due to these benefits the statecharts developed will be used in CIS and many others that will be developed in the future. Although the developed models and integration is not final and more work needs to be done to fully meet all requirements of the CIS. Future work will make statecharts more complex and possibly introduce additional independent statecharts. Many operations will be added to the states of the main declaration statechart that will usually trigger a communication

with an external system (eg. a message to an external system should be sent in ASSESED state to indicate that some type of good has been declared). This type of additions to the statechart make the generated code more complex but generally have little impact to the overall integration between statecharts and client code.

5 Conclusion

In conclusion statecharts proved to be a good way of describing an object's life-cycle based on a customs information system experience. This method solved problems identified with previous solution. Because the code is generated from model the documentation is always up-to-date and corresponds to implementation. Further benefits will become even more evident after reaching maintenance period and implementing modifications to existing system needs to be done. In addition statechart models are easier to develop because it can be done using graphic interface. Statechart formalism offers ways to tackle the state explosion issue, which means that more complexity can be added. Because the used tool is available to everyone interested, the maintenance of the system can be easily passed to another company. This is why previous solution where custom language was used to describe state machines will be replaced with statechart formalism. This thesis provided a detailed overview on how statecharts can be used, developed and integrated to a complex customs information system. These findings can be a valuable input for every development team that has faced similar problems or to teams that have already considered using statecharts in development process.

In future the statechart formalism could be used in other customs information systems. A common library for statecharts should be considered due to the behavioral similarities of these systems. An analysis should be conducted on how the statechart formalism can simplify the maintenance of the system and what type of modifications in the system behavior can be done solely by changing the model in Yakindu. Further, additional technical issues that may arise during the development of the rest of the system need to be addressed.

References

[1] (2019). Enterprise information system (eis) - cio wiki, [Online]. Available: https://cio-wiki.org/wiki/Enterprise_Information_System_(EIS) (visited on 20/12/2019).

[2] R. R. Nelson, "It project management: Infamous failures, classic mistakes, and best practices.", *MIS Quarterly executive*, vol. 6, no. 2, 2007.

[3] T. B. C. Arias, P. Avgeriou, P. America, K. Blom and S. Bachynskyy, "A top-down strategy to reverse architecting execution views for a large and complex software-intensive system: An experience report", *Science of Computer Programming*, vol. 76, no. 12, pp. 1098–1112, 2011.

[4] (2019). Riigikontrolli aruanne riigikogule - avaliku sektori tarkvaraarenduse projektide juhtimine, [Online]. Available: https://www.riigikontroll.ee/DesktopModules/DigiDetail/FileDownloader.aspx?AuditId=2488&FileId=14400 (visited on 20/11/2019).

[5] (2019). Business domains — cybernetica, [Online]. Available: https://cyber.ee/competences/business-domains/#customs-systems (visited on 20/12/2019).

[6] (2019). Electronic customs multi-annual strategic plan, [Online]. Available: https://ec.europa.eu/taxation_customs/sites/taxation/files/resources/documents/customs/policy_issues/e-customs_initiative/masp_strategic_plan_en.pdf (visited on 17/01/2018).

[7] B. Selic, "The pragmatics of model-driven development", *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[8] C. Atkinson and T. Kuhne, "Model-driven development: A metamodeling foundation", *IEEE software*, vol. 20, no. 5, pp. 36–41, 2003.

[9] S. J. Mellor, T. Clark and T. Futagami, "Model-driven development: Guest editors' introduction.", *IEEE software*, vol. 20, no. 5, pp. 14–18, 2003.

[10] M. Samek and L. Quantum Leaps, “A crash course in uml state machines”, *Quantum Leaps, LLC*, 2009.

[11] J. Ganssle, “Embedded state machine implementation”, in *Embedded systems: world class designs*, Citeseer, 2007, ch. 6, pp. 248–257.

[12] G. Kistner and C. Nuernberger, “Developing user interfaces using scxml statecharts”, in *Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML*, 2014, pp. 5–11.

[13] G. Fortino, W. Russo and E. Zimeo, “A statecharts-based software development process for mobile agents”, *Information and Software Technology*, vol. 46, no. 13, pp. 907–921, 2004.

[14] K. R. Leung, L. C. K. Hui, S.-M. Yiu and R. W. Tang, “Modeling web navigation by statechart”, in *Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000*, IEEE, 2000, pp. 41–47.

[15] K. Bogdanov and M. Holcombe, “Statechart testing method for aircraft control systems”, *Software testing, verification and reliability*, vol. 11, no. 1, pp. 39–54, 2001.

[16] R. Swain, V. Panthi, P. K. Behera and D. P. Mohapatra, “Automatic test case generation from uml state chart diagram”, *International Journal of Computer Applications*, vol. 42, no. 7, pp. 26–36, 2012.

[17] A. Topalidou-Kyniazopoulou, “A case (computer-aided software engineering) tool for robot-team behavior-control development”, PhD thesis, Citeseer, 2012.

[18] C. Dragert, J. Kienzle and C. Verbrugge, “Statechart-based ai in practice”, in *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[19] R. Sinha, A. Narula and J. Grundy, “Parametric statecharts: Designing flexible iot apps: Deploying android m-health apps in dynamic smart-homes”, in *Proceedings of the Australasian Computer Science Week Multiconference*, ACM, 2017, p. 28.

[20] C. Guo, Z. Fu, S. Ren, Y. Jiang, M. Rahmaniheris and L. Sha, “Pattern-based statechart modeling approach for medical best practice guidelines-a case study”, in *2017 IEEE 30th International Symposium on Computer-Based Medical Systems (CBMS)*, IEEE, 2017, pp. 117–122.

[21] C. Guo, S. Ren, Y. Jiang, P.-L. Wu, L. Sha and R. B. Berlin, "Transforming medical best practice guidelines to executable and verifiable statechart models", in *Cyber-Physical Systems (ICCPS), 2016 ACM/IEEE 7th International Conference on*, IEEE, 2016, pp. 1–10.

[22] M.-P. Huget, "Generating code for agent uml sequence diagrams", in *Proceedings of Agent Technology and Software Engineering (AgeS)*, 2002.

[23] A. Sadovykh, P. Desfray, B. Elvesæter, A.-J. Berre and E. Landre, "Enterprise architecture modeling with soaml using bmm and bpmn-mda approach in practice", in *2010 6th Central and Eastern European Software Engineering Conference (CEE-SECR)*, IEEE, 2010, pp. 79–85.

[24] (2019). Rational rhapsody - overview - united states, [Online]. Available: https://www.ibm.com/us-en/marketplace/rational-rhapsody (visited on 04/06/2019).

[25] (2019). Magicdraw, [Online]. Available: https://www.nomagic.com/products/magicdraw (visited on 04/06/2019).

[26] (2019). Full lifecycle modeling for business, software and systems — sparx systems, [Online]. Available: https://sparxsystems.com/products/ea/index.html (visited on 15/05/2019).

[27] (2019). Wine - run windows applications on linux, bsd, solaris and macos, [Online]. Available: https://www.winehq.org/ (visited on 22/05/2019).

[28] (2019). Yakindu statechart tools (sct) - state machine tool, [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/ (visited on 21/05/2019).

[29] (2019). Introduction - droolsonboarding, [Online]. Available: https://nheron.gitbooks.io/droolsonboarding/content/ (visited on 04/06/2019).

[30] M. L. Crane and J. Dingel, "Uml vs. classical vs. rhapsody statecharts: Not all models are created equal", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2005, pp. 97–112.

Appendix 1 – Debtor statechart

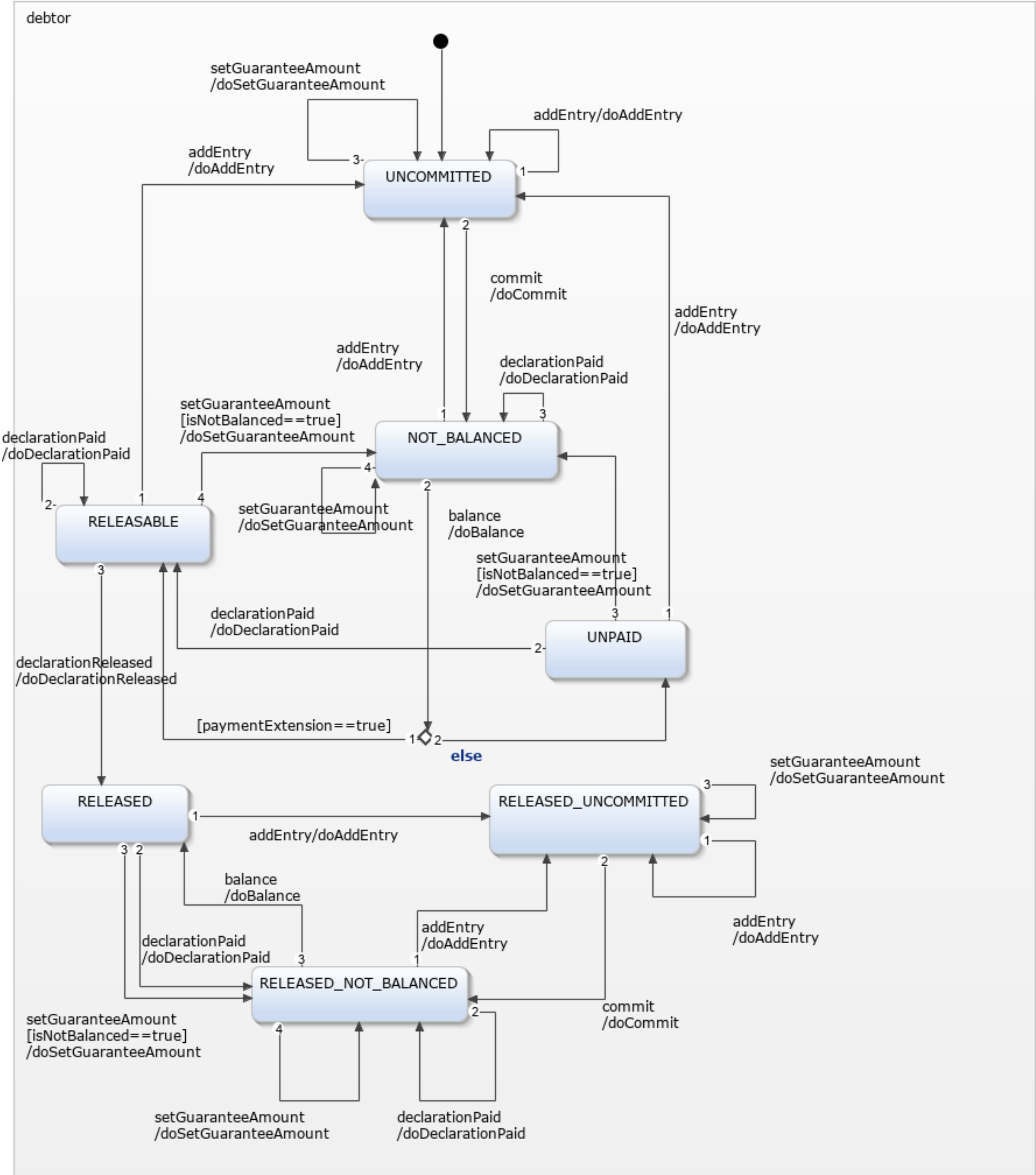

Figure 14. Debtor statechart

Appendix 2 – PersonTaxes statechart

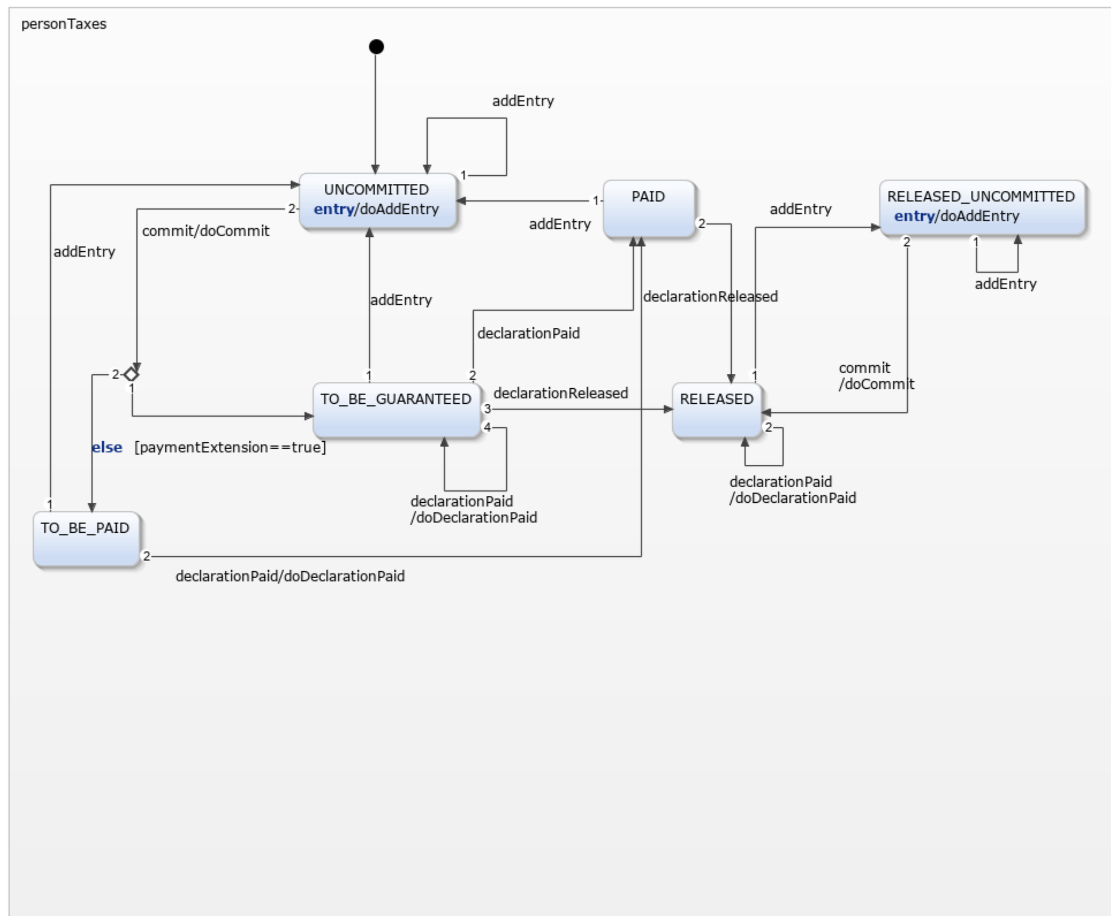

Figure 15. PersonTaxes statechart

Appendix 3 – DebtorStatemachine.java class

```
/** Generated by YAKINDU Statechart Tools code generator. */
package ee.statecharts.debtor;

import ee.statecharts.ITracingListener;
import java.util.LinkedList;
import java.util.List;

public class DebtorStatemachine implements IDebtorStatemachine {
    protected class SCInterfaceImpl implements SCInterface {

        private SCInterfaceOperationCallback operationCallback;

        public void setSCInterfaceOperationCallback(
                SCInterfaceOperationCallback operationCallback) {
            this.operationCallback = operationCallback;
        }
        private boolean addEntry;


        public void raiseAddEntry() {
            addEntry = true;
            runCycle();
        }

        private boolean commit;


        public void raiseCommit() {
            commit = true;
            runCycle();
        }

        private boolean balance;


        public void raiseBalance() {
            balance = true;
            runCycle();
        }
```

```
private boolean declarationPaid;


public void raiseDeclarationPaid() {
    declarationPaid = true;
    runCycle();
}

private boolean declarationReleased;


public void raiseDeclarationReleased() {
    declarationReleased = true;
    runCycle();
}

private boolean setGuaranteeAmount;


public void raiseSetGuaranteeAmount() {
    setGuaranteeAmount = true;
    runCycle();
}

private boolean isNotBalanced;

public boolean getIsNotBalanced() {
    return isNotBalanced;
}

public void setIsNotBalanced(boolean value) {
    this.isNotBalanced = value;
}

private boolean paymentExtension;

public boolean getPaymentExtension() {
    return paymentExtension;
}

public void setPaymentExtension(boolean value) {
    this.paymentExtension = value;
}

protected void clearEvents() {
    addEntry = false;
    commit = false;
```

```java
            balance = false;
            declarationPaid = false;
            declarationReleased = false;
            setGuaranteeAmount = false;
        }
    }


    protected SCInterfaceImpl sCInterface;

    private boolean initialized = false;

    public enum State {
        debtor_UNCOMMITTED,
        debtor_NOT_BALANCED,
        debtor_UNPAID,
        debtor_RELEASABLE,
        debtor_RELEASED,
        debtor_RELEASED_UNCOMMITTED,
        debtor_RELEASED_NOT_BALANCED,
        $NullState$
    };

    private final State[] stateVector = new State[1];

    private int nextStateIndex;

    private List <ITracingListener<State>> ifaceTraceObservers = new LinkedList <ITracingListener<State>>()

    public DebtorStatemachine() {
        sCInterface = new SCInterfaceImpl();
    }

    public void init() {
        this.initialized = true;
        if (this.sCInterface.operationCallback == null) {
            throw new IllegalStateException("Operation␣callback␣for␣interface␣sCInterface␣must␣be␣set.");
        }

        for (int i = 0; i < 1; i++) {
            stateVector[i] = State.$NullState$;
        }
        clearEvents();
        clearOutEvents();
        sCInterface.setIsNotBalanced(false);

        sCInterface.setPaymentExtension(false);
    }
```

```
public void enter() {
    if (!initialized) {
        throw new IllegalStateException(
            "The␣state␣machine␣needs␣to␣be␣initialized␣first␣by␣calling␣the␣init()␣function."
        );
    }
    enterSequence_debtor_default();
}

public void runCycle() {
    if (!initialized)
        throw new IllegalStateException(
                "The␣state␣machine␣needs␣to␣be␣initialized␣first␣by␣calling␣the␣init()␣function.");
    clearOutEvents();
    for (nextStateIndex = 0; nextStateIndex < stateVector.length; nextStateIndex++) {
        switch (stateVector[nextStateIndex]) {
        case debtor_UNCOMMITTED:
            debtor_UNCOMMITTED_react(true);
            break;
        case debtor_NOT_BALANCED:
            debtor_NOT_BALANCED_react(true);
            break;
        case debtor_UNPAID:
            debtor_UNPAID_react(true);
            break;
        case debtor_RELEASABLE:
            debtor_RELEASABLE_react(true);
            break;
        case debtor_RELEASED:
            debtor_RELEASED_react(true);
            break;
        case debtor_RELEASED_UNCOMMITTED:
            debtor_RELEASED_UNCOMMITTED_react(true);
            break;
        case debtor_RELEASED_NOT_BALANCED:
            debtor_RELEASED_NOT_BALANCED_react(true);
            break;
        default:
            // $NullState$
        }
    }
    clearEvents();
}
public void exit() {
    exitSequence_debtor();
}
```

```java
/**
 * @see IStatemachine#isActive()
 */
public boolean isActive() {
    return stateVector[0] != State.$NullState$;
}

/**
* Always returns 'false' since this state machine can never become final.
*
* @see IStatemachine#isFinal()
*/
public boolean isFinal() {
    return false;
}
/**
* This method resets the incoming events (time events included).
*/
protected void clearEvents() {
    sCInterface.clearEvents();
}

/**
* This method resets the outgoing events.
*/
protected void clearOutEvents() {
}

/**
* Returns true if the given state is currently active otherwise false.
*/
public boolean isStateActive(State state) {

    switch (state) {
    case debtor_UNCOMMITTED:
        return stateVector[0] == State.debtor_UNCOMMITTED;
    case debtor_NOT_BALANCED:
        return stateVector[0] == State.debtor_NOT_BALANCED;
    case debtor_UNPAID:
        return stateVector[0] == State.debtor_UNPAID;
    case debtor_RELEASABLE:
        return stateVector[0] == State.debtor_RELEASABLE;
    case debtor_RELEASED:
        return stateVector[0] == State.debtor_RELEASED;
    case debtor_RELEASED_UNCOMMITTED:
        return stateVector[0] == State.debtor_RELEASED_UNCOMMITTED;
    case debtor_RELEASED_NOT_BALANCED:
        return stateVector[0] == State.debtor_RELEASED_NOT_BALANCED;
```

```
    default:
        return false;
    }
}

public SCInterface getSCInterface() {
    return sCInterface;
}

public void addTraceObserver(ITracingListener<State> ifaceTraceObserver) {
    if(!(this.ifaceTraceObservers.contains(ifaceTraceObserver))) {
        this.ifaceTraceObservers.add(ifaceTraceObserver);
    }
}

public void removeTraceObserver(ITracingListener<State> ifaceTraceObserver) {
    if(this.ifaceTraceObservers.contains(ifaceTraceObserver)) {
        this.ifaceTraceObservers.remove(ifaceTraceObserver);
    }
}
public void raiseAddEntry() {
    sCInterface.raiseAddEntry();
}

public void raiseCommit() {
    sCInterface.raiseCommit();
}

public void raiseBalance() {
    sCInterface.raiseBalance();
}

public void raiseDeclarationPaid() {
    sCInterface.raiseDeclarationPaid();
}

public void raiseDeclarationReleased() {
    sCInterface.raiseDeclarationReleased();
}

public void raiseSetGuaranteeAmount() {
    sCInterface.raiseSetGuaranteeAmount();
}

public boolean getIsNotBalanced() {
    return sCInterface.getIsNotBalanced();
}
```

```java
public void setIsNotBalanced(boolean value) {
    sCInterface.setIsNotBalanced(value);
}

public boolean getPaymentExtension() {
    return sCInterface.getPaymentExtension();
}

public void setPaymentExtension(boolean value) {
    sCInterface.setPaymentExtension(value);
}

private boolean check_debtor__choice_0_tr0_tr0() {
    return sCInterface.getPaymentExtension()==true;
}

private void effect_debtor__choice_0_tr0() {
    enterSequence_debtor_RELEASABLE_default();
}

private void effect_debtor__choice_0_tr1() {
    enterSequence_debtor_UNPAID_default();
}

/* 'default' enter sequence for state UNCOMMITTED */
private void enterSequence_debtor_UNCOMMITTED_default() {
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateEntered(State.debtor_UNCOMMITTED);
    }

    nextStateIndex = 0;
    stateVector[0] = State.debtor_UNCOMMITTED;
}

/* 'default' enter sequence for state NOT_BALANCED */
private void enterSequence_debtor_NOT_BALANCED_default() {
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateEntered(State.debtor_NOT_BALANCED);
    }

    nextStateIndex = 0;
    stateVector[0] = State.debtor_NOT_BALANCED;
}

/* 'default' enter sequence for state UNPAID */
private void enterSequence_debtor_UNPAID_default() {
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateEntered(State.debtor_UNPAID);
```

```
    }

    nextStateIndex = 0;
    stateVector[0] = State.debtor_UNPAID;
}

/* 'default' enter sequence for state RELEASABLE */
private void enterSequence_debtor_RELEASABLE_default() {
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateEntered(State.debtor_RELEASABLE);
    }

    nextStateIndex = 0;
    stateVector[0] = State.debtor_RELEASABLE;
}

/* 'default' enter sequence for state RELEASED */
private void enterSequence_debtor_RELEASED_default() {
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateEntered(State.debtor_RELEASED);
    }

    nextStateIndex = 0;
    stateVector[0] = State.debtor_RELEASED;
}

/* 'default' enter sequence for state RELEASED_UNCOMMITTED */
private void enterSequence_debtor_RELEASED_UNCOMMITTED_default() {
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateEntered(State.debtor_RELEASED_UNCOMMITTED);
    }

    nextStateIndex = 0;
    stateVector[0] = State.debtor_RELEASED_UNCOMMITTED;
}

/* 'default' enter sequence for state RELEASED_NOT_BALANCED */
private void enterSequence_debtor_RELEASED_NOT_BALANCED_default() {
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateEntered(State.debtor_RELEASED_NOT_BALANCED);
    }

    nextStateIndex = 0;
    stateVector[0] = State.debtor_RELEASED_NOT_BALANCED;
}

/* 'default' enter sequence for region debtor */
private void enterSequence_debtor_default() {
```

```
    react_debtor__entry_Default();
}

/* Default exit sequence for state UNCOMMITTED */
private void exitSequence_debtor_UNCOMMITTED() {
    nextStateIndex = 0;
    stateVector[0] = State.$NullState$;

    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateExited(State.debtor_UNCOMMITTED);
    }
}

/* Default exit sequence for state NOT_BALANCED */
private void exitSequence_debtor_NOT_BALANCED() {
    nextStateIndex = 0;
    stateVector[0] = State.$NullState$;

    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateExited(State.debtor_NOT_BALANCED);
    }
}

/* Default exit sequence for state UNPAID */
private void exitSequence_debtor_UNPAID() {
    nextStateIndex = 0;
    stateVector[0] = State.$NullState$;

    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateExited(State.debtor_UNPAID);
    }
}

/* Default exit sequence for state RELEASABLE */
private void exitSequence_debtor_RELEASABLE() {
    nextStateIndex = 0;
    stateVector[0] = State.$NullState$;

    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateExited(State.debtor_RELEASABLE);
    }
}

/* Default exit sequence for state RELEASED */
private void exitSequence_debtor_RELEASED() {
    nextStateIndex = 0;
    stateVector[0] = State.$NullState$;
```

```
    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateExited(State.debtor_RELEASED);
    }
}

/* Default exit sequence for state RELEASED_UNCOMMITTED */
private void exitSequence_debtor_RELEASED_UNCOMMITTED() {
    nextStateIndex = 0;
    stateVector[0] = State.$NullState$;

    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateExited(State.debtor_RELEASED_UNCOMMITTED);
    }
}

/* Default exit sequence for state RELEASED_NOT_BALANCED */
private void exitSequence_debtor_RELEASED_NOT_BALANCED() {
    nextStateIndex = 0;
    stateVector[0] = State.$NullState$;

    for(ITracingListener<State> ifaceTraceObserver : ifaceTraceObservers) {
        ifaceTraceObserver.onStateExited(State.debtor_RELEASED_NOT_BALANCED);
    }
}

/* Default exit sequence for region debtor */
private void exitSequence_debtor() {
    switch (stateVector[0]) {
    case debtor_UNCOMMITTED:
        exitSequence_debtor_UNCOMMITTED();
        break;
    case debtor_NOT_BALANCED:
        exitSequence_debtor_NOT_BALANCED();
        break;
    case debtor_UNPAID:
        exitSequence_debtor_UNPAID();
        break;
    case debtor_RELEASABLE:
        exitSequence_debtor_RELEASABLE();
        break;
    case debtor_RELEASED:
        exitSequence_debtor_RELEASED();
        break;
    case debtor_RELEASED_UNCOMMITTED:
        exitSequence_debtor_RELEASED_UNCOMMITTED();
        break;
    case debtor_RELEASED_NOT_BALANCED:
        exitSequence_debtor_RELEASED_NOT_BALANCED();
```

```java
            break;
        default:
            break;
        }
    }

    /* The reactions of state null. */
    private void react_debtor__choice_0() {
        if (check_debtor__choice_0_tr0_tr0()) {
            effect_debtor__choice_0_tr0();
        } else {
            effect_debtor__choice_0_tr1();
        }
    }

    /* Default react sequence for initial entry  */
    private void react_debtor__entry_Default() {
        enterSequence_debtor_UNCOMMITTED_default();
    }

    private boolean react() {
        return false;
    }

    private boolean debtor_UNCOMMITTED_react(boolean try_transition) {
        boolean did_transition = try_transition;

        if (try_transition) {
            if (sCInterface.addEntry) {

                exitSequence_debtor_UNCOMMITTED();
                sCInterface.operationCallback.doAddEntry();


                enterSequence_debtor_UNCOMMITTED_default();
            } else {
                if (sCInterface.commit) {

                    exitSequence_debtor_UNCOMMITTED();
                    sCInterface.operationCallback.doCommit();


                    enterSequence_debtor_NOT_BALANCED_default();
                    react();
                } else {
                    if (sCInterface.setGuaranteeAmount) {

                        exitSequence_debtor_UNCOMMITTED();
```

```
                    sCInterface.operationCallback.doSetGuaranteeAmount();


                    enterSequence_debtor_UNCOMMITTED_default();
                } else {
                    did_transition = false;
                }
            }
        }
    }
    if (did_transition==false) {
        did_transition = react();
    }
    return did_transition;
}

private boolean debtor_NOT_BALANCED_react(boolean try_transition) {
    boolean did_transition = try_transition;

    if (try_transition) {
        if (sCInterface.addEntry) {

            exitSequence_debtor_NOT_BALANCED();
            sCInterface.operationCallback.doAddEntry();


            enterSequence_debtor_UNCOMMITTED_default();
            react();
        } else {
            if (sCInterface.balance) {

                exitSequence_debtor_NOT_BALANCED();
                sCInterface.operationCallback.doBalance();


                react_debtor__choice_0();
            } else {
                if (sCInterface.declarationPaid) {

                    exitSequence_debtor_NOT_BALANCED();
                    sCInterface.operationCallback.doDeclarationPaid();


                    enterSequence_debtor_NOT_BALANCED_default();
                } else {
                    if (sCInterface.setGuaranteeAmount) {

                        exitSequence_debtor_NOT_BALANCED();
```

```
                        sCInterface.operationCallback.doSetGuaranteeAmount();


                        enterSequence_debtor_NOT_BALANCED_default();
                    } else {
                        did_transition = false;
                    }
                }
            }
        }
    }
    if (did_transition==false) {
        did_transition = react();
    }
    return did_transition;
}

private boolean debtor_UNPAID_react(boolean try_transition) {
    boolean did_transition = try_transition;

    if (try_transition) {
        if (sCInterface.addEntry) {

            exitSequence_debtor_UNPAID();
            sCInterface.operationCallback.doAddEntry();


            enterSequence_debtor_UNCOMMITTED_default();
            react();
        } else {
            if (sCInterface.declarationPaid) {

                exitSequence_debtor_UNPAID();
                sCInterface.operationCallback.doDeclarationPaid();


                enterSequence_debtor_RELEASABLE_default();
                react();
            } else {
                if (((sCInterface.setGuaranteeAmount) && (sCInterface.getIsNotBalanced()==true))) {

                    exitSequence_debtor_UNPAID();
                    sCInterface.operationCallback.doSetGuaranteeAmount();


                    enterSequence_debtor_NOT_BALANCED_default();
                    react();
                } else {
```

```
                    did_transition = false;
                }
            }
        }
    }
    if (did_transition==false) {
        did_transition = react();
    }
    return did_transition;
}

private boolean debtor_RELEASABLE_react(boolean try_transition) {
    boolean did_transition = try_transition;

    if (try_transition) {
        if (sCInterface.addEntry) {

            exitSequence_debtor_RELEASABLE();
            sCInterface.operationCallback.doAddEntry();


            enterSequence_debtor_UNCOMMITTED_default();
            react();
        } else {
            if (sCInterface.declarationPaid) {

                exitSequence_debtor_RELEASABLE();
                sCInterface.operationCallback.doDeclarationPaid();


                enterSequence_debtor_RELEASABLE_default();
            } else {
                if (sCInterface.declarationReleased) {

                    exitSequence_debtor_RELEASABLE();
                    sCInterface.operationCallback.doDeclarationReleased();


                    enterSequence_debtor_RELEASED_default();
                    react();
                } else {
                    if (((sCInterface.setGuaranteeAmount) && (sCInterface.getIsNotBalanced()==true))) {

                        exitSequence_debtor_RELEASABLE();
                        sCInterface.operationCallback.doSetGuaranteeAmount();


                        enterSequence_debtor_NOT_BALANCED_default();
```

```
                        react();
                    } else {
                        did_transition = false;
                    }
                }
            }
        }
    }
    if (did_transition==false) {
        did_transition = react();
    }
    return did_transition;
}

private boolean debtor_RELEASED_react(boolean try_transition) {
    boolean did_transition = try_transition;

    if (try_transition) {
        if (sCInterface.addEntry) {

            exitSequence_debtor_RELEASED();
            sCInterface.operationCallback.doAddEntry();


            enterSequence_debtor_RELEASED_UNCOMMITTED_default();
            react();
        } else {
            if (sCInterface.declarationPaid) {

                exitSequence_debtor_RELEASED();
                sCInterface.operationCallback.doDeclarationPaid();


                enterSequence_debtor_RELEASED_NOT_BALANCED_default();
                react();
            } else {
                if (((sCInterface.setGuaranteeAmount) && (sCInterface.getIsNotBalanced()==true))) {

                    exitSequence_debtor_RELEASED();
                    sCInterface.operationCallback.doSetGuaranteeAmount();


                    enterSequence_debtor_RELEASED_NOT_BALANCED_default();
                    react();
                } else {
                    did_transition = false;
                }
            }
```

```
        }
    }
    if (did_transition==false) {
        did_transition = react();
    }
    return did_transition;
}

private boolean debtor_RELEASED_UNCOMMITTED_react(boolean try_transition) {
    boolean did_transition = try_transition;

    if (try_transition) {
        if (sCInterface.addEntry) {

            exitSequence_debtor_RELEASED_UNCOMMITTED();
            sCInterface.operationCallback.doAddEntry();


            enterSequence_debtor_RELEASED_UNCOMMITTED_default();
        } else {
            if (sCInterface.commit) {

                exitSequence_debtor_RELEASED_UNCOMMITTED();
                sCInterface.operationCallback.doCommit();


                enterSequence_debtor_RELEASED_NOT_BALANCED_default();
                react();
            } else {
                if (sCInterface.setGuaranteeAmount) {

                    exitSequence_debtor_RELEASED_UNCOMMITTED();
                    sCInterface.operationCallback.doSetGuaranteeAmount();


                    enterSequence_debtor_RELEASED_UNCOMMITTED_default();
                } else {
                    did_transition = false;
                }
            }
        }
    }
    if (did_transition==false) {
        did_transition = react();
    }
    return did_transition;
}
```

```
private boolean debtor_RELEASED_NOT_BALANCED_react(boolean try_transition) {
    boolean did_transition = try_transition;

    if (try_transition) {
        if (sCInterface.addEntry) {

            exitSequence_debtor_RELEASED_NOT_BALANCED();
            sCInterface.operationCallback.doAddEntry();


            enterSequence_debtor_RELEASED_UNCOMMITTED_default();
            react();
        } else {
            if (sCInterface.declarationPaid) {

                exitSequence_debtor_RELEASED_NOT_BALANCED();
                sCInterface.operationCallback.doDeclarationPaid();


                enterSequence_debtor_RELEASED_NOT_BALANCED_default();
            } else {
                if (sCInterface.balance) {

                    exitSequence_debtor_RELEASED_NOT_BALANCED();
                    sCInterface.operationCallback.doBalance();


                    enterSequence_debtor_RELEASED_default();
                    react();
                } else {
                    if (sCInterface.setGuaranteeAmount) {

                        exitSequence_debtor_RELEASED_NOT_BALANCED();
                        sCInterface.operationCallback.doSetGuaranteeAmount();


                        enterSequence_debtor_RELEASED_NOT_BALANCED_default();
                    } else {
                        did_transition = false;
                    }
                }
            }
        }
    }
    if (did_transition==false) {
        did_transition = react();
    }
    return did_transition;
```

```
        }

    }
```

Appendix 4 – IDebtorStatemachine.java class

```
/** Generated by YAKINDU Statechart Tools code generator. */
package ee.statecharts.debtor;

import ee.statecharts.IStatemachine;

public interface IDebtorStatemachine extends IStatemachine {
    public interface SCInterface {

        public void raiseAddEntry();

        public void raiseCommit();

        public void raiseBalance();

        public void raiseDeclarationPaid();

        public void raiseDeclarationReleased();

        public void raiseSetGuaranteeAmount();

        public boolean getIsNotBalanced();

        public void setIsNotBalanced(boolean value);

        public boolean getPaymentExtension();

        public void setPaymentExtension(boolean value);

        public void setSCInterfaceOperationCallback(SCInterfaceOperationCallback operationCallback);

    }

    public interface SCInterfaceOperationCallback {

        public void doAddEntry();

        public void doCommit();

        public void doBalance();
```

```
        public void doDeclarationPaid();

        public void doDeclarationReleased();

        public void doSetGuaranteeAmount();

    }

    public SCInterface getSCInterface();

}
```

Appendix 5 – IStatemachine.java class

```
/** Generated by YAKINDU Statechart Tools code generator. */
package ee.statecharts;

/**
 * Basic interface for state machines.
 */
public interface IStatemachine {

    /**
     * Initializes the state machine. Used to initialize
       internal variables etc.
     */
    public void init();

    /**
     * Enters the state machine. Sets the state machine
       into a defined state.
     */
    public void enter();

    /**
     * Exits the state machine. Leaves the state machine
       with a defined state.
     */
    public void exit();

    /**
     * Checks whether the state machine is active.
     * A state machine is active if it has been entered.
       It is inactive if it has not been entered at all
    or if it has been exited.
     */
    public boolean isActive();

    /**
     * Checks whether all active states are final.
     * If there are no active states then the state machine
       is considered being incative. In this case this
       method returns <code>false</code>.
```

```
     */
    public boolean isFinal();

    /**
    * Start a run-to-completion cycle.
    */
    public void runCycle();
}
```

Appendix 6 – ITracingListener.java class

```
/** Generated by YAKINDU Statechart Tools code generator. */
package ee.statecharts;

/**
 * Tracing interface for state machines.
 * Must be implemented if tracing feature is used.
 * Multiple listeners can be added to and removed
    from the state machine.
 */
public interface ITracingListener<T> {

    /**
     * OnStateEntered will be called if any state
        has been entered.
     */
    void onStateEntered(T state);

    /**
    * OnStateExited will be called if any state
        has been exited.
    */
    void onStateExited(T state);
}
```