MASTER'S THESIS

Student Interaction Module

Reusable architecture for the front-end of an Intelligent Tutoring System

Zijlstra, C.

Award date:
2021

Link to publication

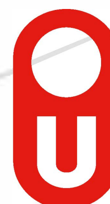
General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

Take down policy
If you believe that this document breaches copyright please contact us at:
pure-support@ou.nl
providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 12. Dec. 2021

Open Universiteit
www.ou.nl



C. (Cor) Zijlstra

Student Interaction Module

Reusable architecture for the front-end of an Intelligent Tutoring System

Thesis presentation: 26 February 2021

Web

Student Interaction Module

Reusable architecture for the front-end of an Intelligent Tutoring System

by

C. (Cor) Zijlstra

in partial fulfilment of the requirements for the degree of

Master of Science

in Software Engineering

at the Open University of the Netherlands, Faculty of Science

Master's Programme in Software Engineering

to be defended publicly on 26 February 2021

Student number:

Course code: IM9906

Thesis committee: Dr. B. Heeren (supervisor), Open University of the Netherlands

Dr. J. Lodder , Open University of the Netherlands

Dr. Ir. S. Stuurman, Open University of the Netherlands

Contents

Abstract iii

1. Introduction 1

2. Research 2

   2.1. Research questions 2

   2.2. Research method 3

3. Research context 4

   3.1. Different architectures of an ITS 4

      3.1.1. Traditional or four-model architecture 4

      3.1.2. Other architectures 5

      3.1.3. The integration-oriented architecture 5

   3.2. The functionality of an ITS 6

      3.2.1. Tasks of a human tutor 6

      3.2.2. Tasks of an ITS 6

      3.2.3. Tasks of a logic ITS 6

   3.3. Ideas framework 7

   3.4. Related work 8

4. Requirements 10

   4.1. Stakeholders 10

   4.2. Functional requirements 10

      4.2.1. How does an ITS work? 11

      4.2.2. Based on an interview 11

      4.2.3. Tasks of an ITS in theory 12

      4.2.4. A further elaboration 13

   4.3. Non-functional requirements of the Student Interaction Module 16

5. The SIM Architecture 19

   5.1. Architectural drivers 19

   5.2. Trade-offs 19

      5.2.1. Trade-offs between the requirements 19

      5.2.2. Architecture design trade-offs 20

   5.3. Architecture views 25

      5.3.1. Entity view 25

      5.3.2. Overall view 27

      5.3.3. Functional view 28

      5.3.4. Development view 35

6. Proof of Concept ... 43
6.1. General flow of the program ... 43
6.1.1. Structure of the program ... 43
6.1.2. Variables and type definitions ... 44
6.1.3. Retrieving data ... 44
6.1.4. Feedback and messages ... 45
6.2. Modules ... 46
6.2.1. Overall view of the proof of concept ... 46
6.2.2. Type definitions in the Model modules ... 47
6.2.3. The view modules ... 49
6.2.4. Main ... 50
6.2.5. Exercise selection ... 51
6.2.6. Answer ... 52
6.2.7. Hint ... 53
6.2.8. Rule ... 54
6.2.9. Feedback ... 55
6.2.10. Communication ... 56
6.3. Architecture vs proof of concept ... 57
7. Flexibility of the architecture ... 62
8. Discussion ... 66
8.1. Limitations of the study ... 66
9. Conclusions and recommendations ... 68
9.1. Conclusions ... 68
9.2. Future work ... 69
Appendix ... 72
A. Adding other functionality ... 72
A.1. Exercise selection ... 72
A.2. Answer ... 73
A.3. Rule ... 75
A.4. Hint ... 75
A.5. Feedback ... 77
A.6. Communication ... 78

Abstract

Although working from home has become increasingly important, since the outbreak of the COVID-19 virus, almost the entire world has learned even more the importance of being able to work from home. Many people had to stay at home because of the lockdowns. This also applies to schools; teachers and students had to stay at home and schools had to offer alternatives to teach the students.

Intelligent Tutoring Systems (ITSs), systems with which students can learn learning material from a computer, already existed before those days but can contribute in those needs. A big problem with these ITSs is the cost of development and maintenance of the software.

This cost of development can possible be reduced when parts of an ITS can be used by another ITS. In this study we examine a possible architecture of the front-end of a logic ITS, by determining the requirements, creating an architecture based on those requirements and making a proof of concept of the front-end. The studied architecture is based on a product line architecture, where components are reusable in multiple versions of the front-end, but it is recommended to add an extra entity related to the connected module. Besides that the relation between feedback and hints in relation to the SIM architecture needs to be studied.

When the front-end or parts of it are reusable, it would reduce the development costs of an ITS, which would improve usability for education.

1. Introduction

An Intelligent Tutoring System (ITS) is a system to support education by teachers (not to replace them) using AI-technology. There are many ITSs in support of all types of education, such as mathematics, geography, computer programming, languages and chemistry. Much research has been done on ITSs and especially their AI part, but ITSs are not often used to support education. One of the reasons for this is that they are expensive to develop; according to Murray (1999) the cost of development of 1 hour learning in an ITS is 300 hours. Although new developments since Murray's publication have resulted in shorter development times, the cost of development are still high.

ITSs are systems that support tutoring by means of AI-technology. The AI-technology is used to simulate a teacher who knows what, to whom in which way has to be learned (Nwana, 1990). Tutoring systems are also known as Intelligent Computer Aided Instruction (ICAI). ICAI is a successor of CAI (Computer Aided Instruction) which are systems that offer a standard learning program according to programmed rules, where each student always has the same subject matter in the same way (Wenger, 1987). In addition to imparting certain knowledge to students, researching study processes, such as when are what kind of hints the most effective, is a goal of an ITS (Anderson, 1987).

Like most computer systems ITSs are divided into modules, each of which provides its functionality to the ITS. Since there are no real standards for developing an ITS, there are many different configurations of such modules. Even if the ITSs are composed of similar modules, the communication between those modules may differ. This makes it difficult to use modules from other ITSs.

In this study we examine the architecture of a frond-end for ITSs that can communicate with modules of different ITSs. We concentrate us on a reusable or partly reusable frond-end for a logic ITS that is part of a traditional or four model architecture. In this study, we call this front-end the Student Interaction Module (SIM.

The architecture of this SIM can be used to develop front-ends for existing modules of ITSs and for newly developed modules of ITSs. Due to its reusability, a cost reduction can be achieved. This cost reduction can contribute in the usage of ITSs.

In chapter 2 we describe the research questions and the research method with the validations that are used. In chapter 3 we give the context of the SIM and look at some architectures of ITSs. In this chapter we also look at the functionality of the modules of a traditional four-model architecture and we briefly describe the IDEAS framework, which we used for our study. In chapter 4 we look at the stakeholders of the SIM and the requirements those stakeholders have for the SIM. In chapter 5 we describe the architecture of the SIM, but before that we first give the architecture drivers and the most important trade-offs that were important in the development of this architecture. To prove that a SIM can be built with the architecture we have made a proof of concept. Information about this proof of concept with the differences from the architecture can be found in chapter 6. In chapter 7 we look at the possibilities to use the architecture for other ITSs. In chapter 8 the conclusions and future work are described.

2. Research

In this chapter we describe the research questions and the research method we have used.

2.1. Research questions

In this study we concentrate us on the architecture of the SIM

An architecture of a software system depends on functional and non-functional requirements of the stakeholders of the software system. These requirements can conflict with each other, and for those situations compromises or trade-offs have to be found, before an architecture can be proposed.

Goal of the study

Our goal is to examine an architecture for the SIM. This architecture can be used to build reusable parts for SIMs. There are many different ITSs with different purposes and we limit our goal to: the SIMs should at least be able to support those ITSs that support stepwise logic exercises and are based on a traditional four model architecture or a combination of modules from different ITSs with a four module architecture or a Learning Environment (LE). This study focuses on ITSs that expect answers through formulas: more graphic answers such as semantic tableaux are not taken into account. The reuse of the SIMs or parts of those SIMs must reduce the development time and cost of a SIM.

RQ 1

The first research question (RQ 1.) is: what are the requirements for the SIM? Before we can make an architecture we need to know which functionalities it should be able to support and which actions it should take given certain conditions (the functional requirements) and which other qualities the SIM should have (the non-functional requirements). These requirements come from other studies and an interview with a stakeholder.

RQ 2

The second research question (RQ 2.) is: what are the trade-offs for these requirements? The requirements of the stakeholders can conflict with each other. In those situations we have to find compromises between them. These compromises are the trade-offs of the SIM.

With the requirements and the trade-offs we can make an architecture of the SIM. During this step we have to make design decisions, which also can lead to trade-offs.

RQ 3

The third research question (RQ 3.) is: what is the flexibility of the architecture? For this study, there are too many ITSs to make an architecture that will fit for every ITS. By looking at the flexibility of our architecture we want to explore the possibilities to add extra functionality to our architecture in order to connect modules of ITSs that cannot be connected to our current architecture.

2.2. Research method

In this paragraph the research method is described with the related chapters in this study.

RQ 1.

For the answering of RQ 1 we have done a literature study to find possible functionalities and qualities of the SIM for a traditional four model architecture of a stepwise based ITS and we have had an interview with dr. J. Lodder from the Open University of The Netherlands. She is a teacher and researcher of logic courses and also a developer of ITSs using the IDEAS Framework.

The interview with Dr. J. Lodder was done to validate whether the found functionalities and qualities were complete. This resulted in some extra points for attention for the SIM.

RQ 1 is handled in chapter 4 of this study.

RQ 2.

For the answering of RQ 2 we determine whether there are conflicts between requirements for which we have to make trade-offs. Next we design an architecture for the SIM, during this we have to make design-decisions that also can result in trade-offs. For the designing of the architecture we will use literature about how to make an architecture and look at studies about the architecture of ITSs. RQ 2 is handled in chapter 5 of this study.

To validate if the SIM architecture can be used to build a SIM a prototype of the SIM has been made. Elmex, an experimental LE, that works with the IDEAS Framework, is used as a base for this. This proof of concept is described in chapter 6 of this study.

RQ 3.

To answer RQ 3, we look at two ITSs for topics other than logic, which clearly deviate from the investigated logical ITSs, and describe whether and how the SIM can be implemented for this purpose with the architecture. We cannot check if it is possible to connect a SIM based on our architecture to every logic ITS that is based on a four model architecture, because there are too many and it is likely that new logic ITSs will be developed. By looking at non-logic ITSs and describing to what extent the SIM can be developed with our architecture we want to show the flexibility of our architecture. If the architecture is flexible enough to support non-logic ITSs it is more likely that it also can support more logic ITSs based on a four model architecture. RQ 3 is handled in chapter 7 of this study.

3. Research context

In this chapter we describe what an ITS is by looking at the architecture and at the functionality of it and we also look at the IDEAS framework that we will use for our proof of concept.

3.1. Different architectures of an ITS

In this paragraph we look at some architectures of ITSs.

3.1.1. Traditional or four-model architecture

There is no set description on how an ITS should be structured, but the most common layout consists of the following four parts (see Figure 1) (Nwana, 1990; Padayachee, 2002)

- The expert knowledge module
- The student model module
- The tutoring module
- The user interface module

Below is a brief summary of the functionality per module. It should be noted that the communication between the various modules from a software architecture point of view is not clearly described (Heeren and Jeuring, 2014).

Expert knowledge module

The expert knowledge module (also known as the domain model module or domain reasoner module) has two tasks: on the one hand it contains the curriculum to be taught to a student, and on the other hand it contains information about how the student's knowledge should be assessed in response to given answers and reactions (Goguadze, 2009).

Student model module

The student model module is the heart of an ITS (Nkambou et al., 2010). It contains all information about the students, the knowledge of the students and their learning progress (Brusilovskiy, 1994).

Tutoring model module

The tutoring model module is responsible for the pedagogical part of an ITS. It uses information from the student model module and it determines how the knowledge from the expert knowledge module should be taught to the student.

User interface module

The user interface module takes care of the communication between the ITS and the student and is responsible for how the information from the ITS is presented to the student and how the input of the student is processed.

We think that Student Interaction Module (SIM) is a better term for this module, because it emphasizes the interaction between the student and the system. A user interface is more a device connected to the system which receives input from a user by means of an input device like a keyboard, mouse or a microphone, or produces output through an output device like a screen or a speaker.

In our study we use the term Student Interaction Module, that is a piece of software that handles the input from the input devices and transforms those to output for the output devices and if necessary to the connected modules of the ITS, so that the student can interact with the ITS.

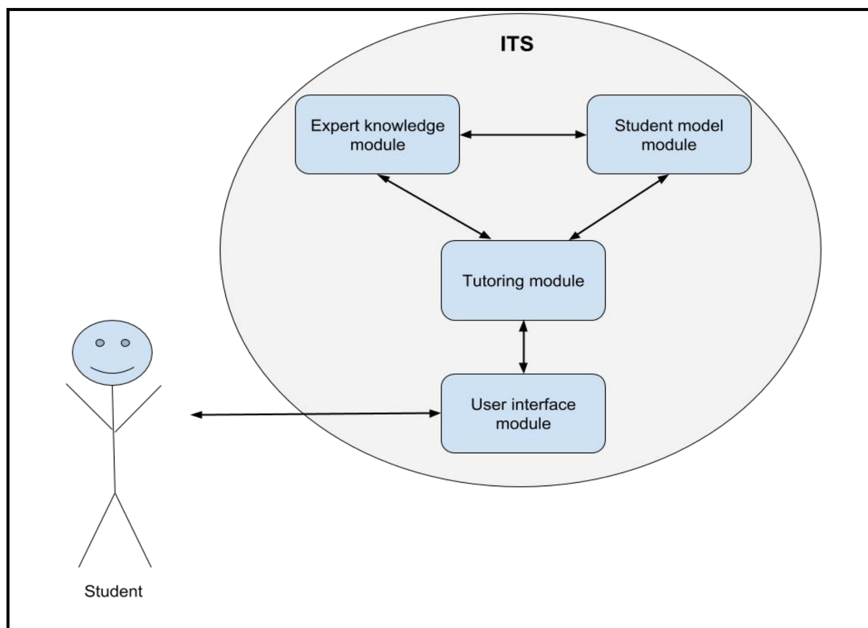


FIGURE 1. TRADITIONAL ARCHITECTURE OF AN ITS (NWANA, 1990)

3.1.2. Other architectures

Nwana mentions in her study (1990) several other architectures such as Anderson's Advanced Computer Tutoring (ACT) ITS architecture (Anderson et al., 1985a, 1985b), the Hartley & Sleeman architecture (Hartley and Sleeman, 1973), the O'Shea et al. architecture (O'Shea et al., 1984) and the self-improving architecture (Kimball, 1982; O'Shea, 1979). These are architectures that concentrate on a part of the possible functionality of an ITS (Nkambou et al., 2010; Nwana, 1990).

Padayachee mentions in her study (2002) the three-model architecture. This architecture is the same as the four-model architecture except for the user-interface, which is not seen as a part of the ITS system in that architecture. She also mentions the Self architecture (Self, 1998) which can be seen as an extended version of the three-model architecture and she mentions the Siemer's & Angelides' general intelligent tutoring system architecture which is the same as the three-model architecture with an extra part called the overall system control which is responsible for checking the operations of the ITS (it checks what the ITS does versus what should the ITS do?) (Siemer and Angelides, 1998).

There are also other architectures like for example ActiveMath where there is an exercise subsystem which communicates with the domain reasoner, the student model, the tutorial component and the user interface (Goguadze, 2009).

There are also ITSs that only have an expert knowledge module and a user interface module. Lodder et al. (2016) indicate that most ITSs for propositional logic have such an architecture and call those Learning environments (LEs). Although there is no tutoring module and no student module in this architecture, seen from the user interface module or SIM as we call it, this architecture looks a lot like the traditional or four-module architecture; the user interface module receives learning material for the student, but the contents of it is not prepared for that particular student.

3.1.3. The integration-oriented architecture

Originally the technical architecture of an ITS was based on one system which was located at one location. Brusilovsky (1995) introduced an integration-oriented architecture in which the components of an ITS could be replaced by other components of other ITSs with a similar architecture. Ritter and Koedinger (1996) give with the plug-in architecture an implementation of that integration-oriented architecture and Ritter et al. (1998) describe a

component-based system which is also an implementation of it. Heeren and Jeuring (2014) did a study on feedback services which is a refinement of the plug-in and the component based architecture, in which they proposed to use web-services for the communication with expert knowledge modules.

The ideas of these studies can be combined to an ITS which exists of modules of more ITSs. Examples of such ITSs are ActiveMath that can communicate with several domain reasoners (Goguadze, 2009) and the study of Aleven et al. (2017) in which they integrated GIFT (Sottilare et al., 2012) and CTAT (Aleven et al., 2016) into the edX Massive Open Online Course (MOOC) platform.

3.2. The functionality of an ITS

In this paragraph we look at the required functionality of a user interface of an ITS. In order to do this, we start by looking at the tasks of a human tutor. Next we compare those tasks with the ones of an ITS based on a traditional architecture. Because we are interested in a logic ITS the next step is looking at the tasks of a logic ITS.

3.2.1. Tasks of a human tutor

Ritter et al. (1998) determine the functionality of an ITS by looking at the tasks of a human tutor. A human tutor explains the material that has to be learned and tells how that knowledge can be used to solve problems. He also gives examples related to the material and selects based on experience and knowledge about the learning objectives and experience of the students about problems to be solved. When the student is solving a problem, the tutor gives hints and feedback. Feedback is also given after analysing the answer on a problem. An ITS is complete if it performs all of these tasks (Ritter et al., 1998).

3.2.2. Tasks of an ITS

In his study VanLehn (2006) compares six different ITSs in terms of how they are functioning and mentions possible tasks of an ITS. Compared to the tasks of the human tutor there are two differences:

VanLehn also mentions the possibility that students can choose their own problems to solve. This is an extension of the tasks of the human tutor.

VanLehn states that the ITS determines whether a student needs a hint. This is more precisely formulated than mentioned by the tasks of a human tutor but in the end the human teacher decides for himself or herself whether to give a hint about the exercise.

Concluding we can say that the possible tasks VanLehn mentions are the same as those mentioned by Ritter et al. There is only one extra possible task: offering the students the possibility to choose their own problems to solve. This means that the possible tasks of an ITS that VanLehn mentions can be used to define the functionality of an ITS.

3.2.3. Tasks of a logic ITS

In their research into a domain reasoner for propositional logic, Lodder et al. (2016) compare six Learning Environments for teaching logic and they recognize possible tasks of such a Learning Environment (see Table 1).

Compared to the tasks of an ITS from the previous paragraph the tasks are more specific on the subject of logic and the tasks related to the student module are not mentioned.

Possible tasks of a logic LE (Lodder et al., 2016)
Offer different kind of tasks (determine normal form or prove an equivalence)
Support various types of exercises (a fixed set of exercises, random generated exercises or user-entered exercises)
Support various types of answers (only a formula, only a rule to apply or both)
Support the possibility to work in two directions when proving an equivalence.
Support several kinds of feedback (only right/wrong, a description of the right answer, a description of the error-made, Information about how to proceed).
Support several kinds of feedforward (giving a hint for the next step to take, explicitly indicate next step, global description to use components for a next step. In addition, feedforward can also be given in one or two directions if it is possible to elaborate two directions)
Give solutions to problems and/or give examples of solutions.
Support the possibility to change the set of rules, notation mode and strategy which can be used to solve problems.

TABLE 1: POSSIBLE TASKS OF A LOGIC ITS

3.3. Ideas framework

The IDEAS framework is a framework for developing domain reasoners that give intelligent feedback[1]. Logic tutors LogEX [2], LogAx[3], among others, are developed using this framework. These ITSs are learning environments.

LogEX (see: Figure 2) has three kind of exercises: convert to disjunctive normal form, convert to conjunctive normal form and prove logical equivalence. With LogAx (see: Figure 3) you can practice axiomatic proof. Both have stepwise-exercises where you have to enter a formula and the rule that has been applied. You can ask for hints in both, part of the solution or the complete solution, these options can be turned on and off through configuration settings. Feedback is provided after entering a solution step in both. In LogEX when proving logical equivalence there is the possibility to enter steps top-down and bottom up. In LogAx the formulas can be assigned step numbers, so that other formulas can refer to those step numbers.

[1] https://www.uu.nl/en/research/software-systems/software-technology-for-learning-and-teaching/research-themes/the-ideas-project
[2] http://IDEAS.cs.uu.nl/logex/
[3] http://IDEAS.cs.uu.nl/logax/

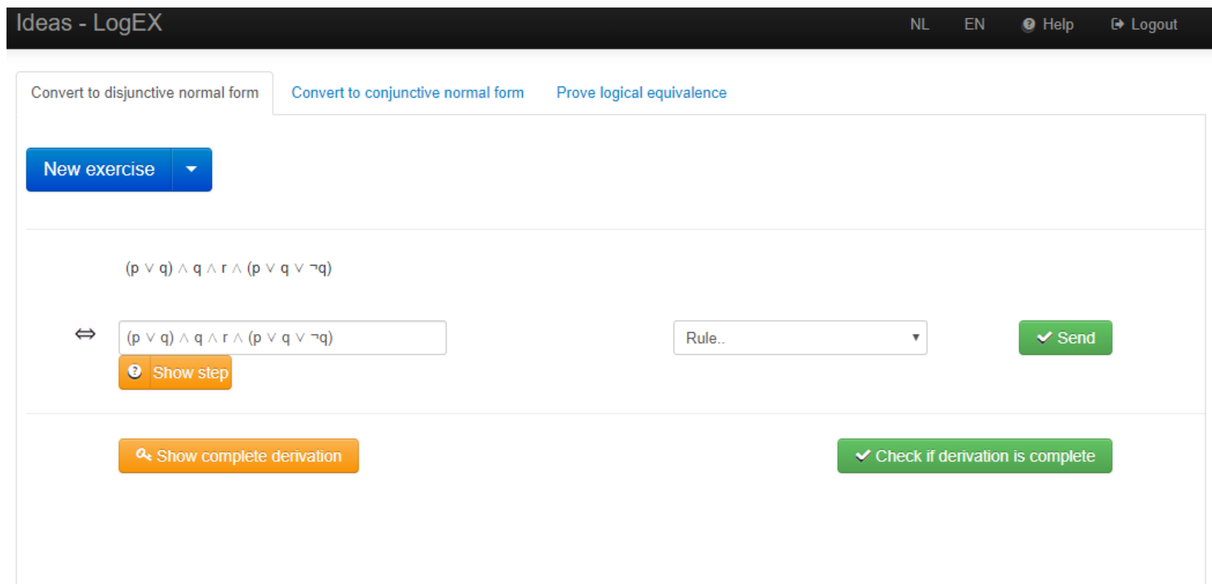


FIGURE 2: LOGEX

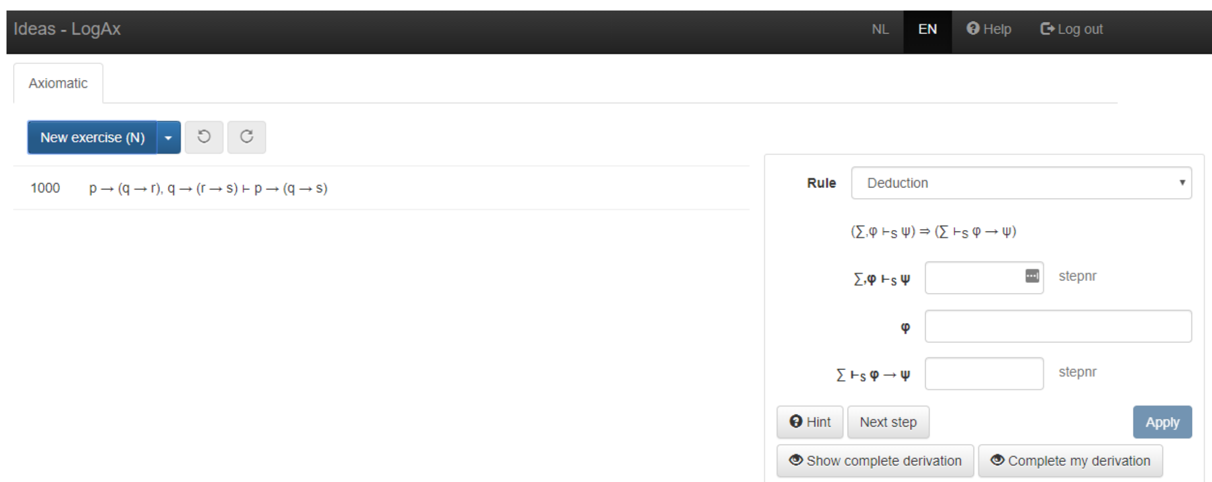


FIGURE 3: LOGAX

3.4. Related work

In this paragraph some other studies related to our study are mentioned, and the differences with our study are explained. We have looked for studies about the functionality and architecture of a frond-end of an ITS and studies about connecting the frond-end to modules of different ITSs.

The integration-oriented architecture is an architecture in which components of an ITS can be replaced by similar components of another ITS (Brusilovsky, 1995). This idea is the basis of our study; it describes an ITS architecture in which components can be replaced by similar ones. We describe the architecture of the SIM, which is one of the components of a four model ITS architecture. The SIM should be able to connect to different other components like Brusilovsky described.

In Ritter and Koedinger (1996) Microsoft Excel is used as a front-end for mathematic exercises supplied by ITSs and they use a translator between the Excel front-end and the connected modules. This solution does not use internet, but is installed completely on the user's device. Ritter and Koedinger mention some ideas on possible solutions using internet, but these ideas are not implemented. Excel is in the study of Ritter and Koedinger the front-end that can be connected to modules of different ITSs, but in this study the functional requirements and the architecture of the front-end are not examined; they describe an architecture based on the

usage of Excel. In our study we start with determining the requirements for the frond-end and describe an architecture for the frond-end based on those requirements.

Alpert et al. (1999) give in their study some non-functional requirements for a user interface of an ITS. They state that the user interface should communicate through a web server, in order to make it possible to work location and time independent, which makes it possible that more students can make use of the ITS. In order to make this possible it is needed that the complete system has a short response-time. This study does not describe the functional requirements for the frond-end, like our study does.

Studies such as Goguadze (2010) and Heeren and Jeuring (2014) mention the front-end as part of the architecture, but those studies concentrate on the architecture of the complete ITS and especially the other components within such an ITS.

4. Requirements

In the previous chapter we have described what an ITS is by looking at the architecture and the functionality of it. In our study we concentrate on the user interface module of a traditional or four-module architecture that we call the Student Interaction Module (SIM).

In this chapter we will determine the stakeholders of the SIM and the functional and non-functional requirements for the SIM; the answer on RQ 1 of our study. Normally the requirements of a system are based on the concerns of the stakeholders, but we have decided to determine the requirements by using existing studies. Hence, we only describe the concerns of the stakeholders in broad terms.

4.1. Stakeholders

Before we look at the requirements that the SIM should meet, we look at the stakeholders (the parties that are involved). These are divided in primary stakeholders, secondary stakeholders and indirect users, who all have their own concerns for the SIM.

Primary stakeholders

The student is a primary stakeholder of the SIM. The student wants to learn the learning material by making exercises, getting understandable hints and feedback. The student does not want to be bound to a special place or a specific time to do the learning.

Secondary stakeholders

Teachers are secondary stakeholders. The teachers need a system that can support them teaching learning material to the students. The ITS should offer a broad set of tasks and exercise tailored to student needs. For teachers it is needed that the ITS supports adjustability when it comes to whether or not to give hints and feedback and which logic rules may be used at that time.

Researchers are secondary stakeholders. Researchers need a front-end that can be connected to modules or scenarios they are researching. Those scenarios can for example be related to whether or not to give hints and feedback and which logic rules may be used at that time.

Developers are secondary stakeholders. For developers it is important that the software is easy to maintain and because other modules have to be added, the software should also be easy to extend.

System administrators are also secondary stakeholders. System administrators want to have a high availability of the software and little work needed for running the software. They also want to have freedom in choosing the platform on which the software is used.

Indirect user

The educational institutes are the other stakeholders. The educational institutions want to have a good reputation for the education that is offered at the lowest possible cost.

4.2. Functional requirements

In this paragraph we describe the functional requirements of the SIM. First we describe how an ITS could work, next we give the important points from an interview with dr. J. Lodder, afterwards we describe the tasks of an ITS based on a literature study and finally we give a further elaboration on some of the requirements.

4.2.1. How does an ITS work?

Before we look at the functional requirements of a logic ITS we first describe how an ITS could work.

With the four-module architecture a student gives his credentials in the user interface module. These credentials are received by the tutoring module that forwards those to the student model module. The student model module has knowledge about the student, it knows the progress of the student and the learning capabilities of the student. This information is sent back to the tutoring module. The tutoring module retrieves a list of possible tasks and exercises from the expert knowledge module and selects an exercise for the student, which is sent to the user interface module to be solved by the student. In addition to the exercise, the knowledge expert module may also have options such as giving hints, giving example exercises, feedback on answers and possible solution strategies. The tutoring module determines which of the available possibilities of the expert knowledge module are available in the user interface module. Next the student interacts with the user interface module, he gives an answer, or, depending of the possibilities, he asks for a hint or an example. The action of the student is sent to the tutoring module, which forwards it to the expert knowledge module. The expert knowledge module returns feedback to the tutoring module. The tutoring module determines next how this reaction is sent back to the user interface module. Meanwhile the tutoring module also sends information to the student module, so that this module can update its information about the student. When a student has finished an exercise the tutoring module selects the next exercise for the student.

There are also other possibilities. For example the logic learning environments LogEX and LogAx (see paragraph 3.3) do not have a tutoring and a student model module (Lodder et al., 2016). In this case the ITS does not have information about the student and there is not a tutoring module that can control the exercises that are presented to the student (the so called "outerloop" of an ITS) nor the steps when solving an exercise (the so called "innerloop" of an ITS). In this situation it can be needed that this is done by the SIM.

4.2.2. Based on an interview

On October third 2019 we had an interview with dr. J. Lodder from the Open University of the Netherlands. She is a teacher and researcher of logic on this university and a developer of some of the LEs based on the IDEAS framework. Because of her knowledge on the subject of logic and the concerns of the stakeholders of the front-end of a logic ITS we interviewed her about the important aspects of a SIM. Below the additional points dr. J. Lodder mentioned, in paragraph 4.2.3 we discuss the normal tasks of an ITS. LEs without a student module can be created on the IDEAS Framework. The inner and the outer loop are controlled by the user interface in these LEs.

1. The ITSs within the IDEAS framework can be used as a teaching environment in which students can solve exercises and use freely the options for feedback, hints and so on, but it can also be used as a testing environment for the teacher in which the available options and the exercises are controlled by the teacher. In this case the teacher can decide which exercises are to be solved by the students and which and when feedback and hints are given. Because there is not a student module in the ITS all these options should be controlled by the SIM.
2. With the SIM it must be possible to translate and transform messages and formulas in a way that is understandable for the students on the one hand, but can also be used by the connected module. Messages that need to be translated or transformed are all kinds of hints and feedback given by the connected module. These may be formulated in a way that is not understandable to the student, for example in another language or with incomprehensible codes (like logic.propositional.notnot in the IDEAS framework). Those should be presented in a way that is recognized by the student, this may differ from the

way of communicating with the connected module. (Remark: Translation and transformation of hints and feedback is, when using a complete ITS, a task of the tutoring module, but in case of an LE there is no tutoring module and there is also the possibility that the used tutoring module cannot be changed. This option can be used in those situations).

4.2.3. Tasks of an ITS in theory

The responsibility of the SIM is taking care of the communication between the student and the rest of the ITS. This means that the SIM should support all the possible tasks of a logic ITS from paragraph 3.2.3 and the tasks related to the tutoring module and the student model module when those are related to the communication with the student.

Offer different kind of tasks – Different kind of tasks are: a fixed list of exercises, exercises which are made randomly and exercises entered by the student (Lodder et al., 2016) or a task selected by the student from a list of tasks (VanLehn, 2006). The SIM should be able to support all these kinds of exercises.

Support various types of exercises – The SIM should support the types of exercises that are offered by the connected ITS modules. For example it should be able to support rewriting in disjunctive normal form (DNF) or conjunctive normal form (CNF) and proving equivalence (Lodder et al., 2016).

Support various types of answers –The SIM has to support various types of answers. Those answers can be for example only formulas or only applied rules, or formulas and rules (Lodder et al., 2016).

Support the possibility to work in two directions when proving an equivalence – The SIM should support this, when this is supported by the rest of the connected ITS components (Lodder et al., 2016).

Support several kinds of feedback – The SIM should be able to show feedback as it is provided by the rest of the ITS. Feedback can be given after a step or after completing the exercise (VanLehn, 2006). In addition, it should be possible to use special characters like a check mark and different colours like red for incorrect and green for correct.

Support several kinds of feedforward – Feedforward or hints can be given by the rest of the ITS. The SIM should be able to show those. Depending on the connected modules there can be several kinds of feedforward which the SIM should support.

Feedforward can be given on request of the student (VanLehn, 2006), when this is possible there must be a possibility for the student to request this. Feedforward can also be given after a certain time or a defined number of wrong answers (VanLehn, 2006). In case it is possible to work in two directions feedforward can also be given in two directions, the SIM should support this (Lodder et al., 2016).

Give solutions to problems and/or give examples of solutions – If the connected modules of the ITS support these options the SIM should support them as well.

Support the possibility to change the set of rules, notation mode and strategy that can be used to solve problems – ITSs can have different sets of rules, notation models and strategies to solve problems. The SIM should be able to support those possibilities.

4.2.4. A further elaboration

In this paragraph we look at a couple of items from the last two paragraphs for a further elaboration.

Feedback and feedforward

Feedback and feedforward are related to each other, that is why we discuss them here together. We use feedback for the result after checking an answer on an exercise or a step in the answer and we use feedforward or hints as a response on a request of a student for help. Feedback can also contain a hint, but we only use hints in our study when it is on request of the student.

Feedback and feedforward can have the following kinds of types: knowledge of performance, knowledge of result/response, knowledge of the correct response, answer-until-correct, multiple-try feedback and elaborated feedback (Narciss, 2008). What types of feedback and feedforward are available depends on the connected module and the preferences of the teacher.

Knowledge of performance (KP) is about the performance of the student for a set of tasks. Knowledge of result/response (KR) is for a step or an exercise; correct or incorrect. Knowledge of the correct response (KCR) gives the correct answer for an exercise. Answer-until-correct (AUC) is combined with KR and is given until the answer is correct. Multiple-try feedback (MTF) is also combined with KR and can be limited to the number of tries the student has. Elaborated feedback (EF) gives extra information with KR and KCR and is divided in: Knowledge about Task Constraints (KTC), Knowledge about Concepts (KC), Knowledge about Mistakes (KM), Knowledge about How to proceed (KH), and Knowledge about MetaCognition (KMC) (Narciss, 2008). See Table 2 for the examples of these types of elaborated feedback.

Knowledge about task constraints (KTC)	Hints/explanations on type of task
	Hints/explanations on task-processing rules
	Hints/explanations on subtasks
	Hints/explanations on task requirements
Knowledge about concepts (KC)	Hints/explanations on technical terms
	Examples illustrating the concept
	Hints/explanations on the conceptual context
	Hints/explanations on concept attributes
	Attribute-isolation examples
Knowledge about mistakes (KM)	Number of mistakes
	Location of mistakes
	Hints/explanations on type of errors
	Hints/explanations on sources of errors
Knowledge about how to proceed (KH)	Bug-related hints for error correction
	Hints/explanations on task-specific strategies
	Hints/explanations on task-processing steps
	Guiding questions
	Worked-out examples
Knowledge about metacognition (KMC)	Hints/explanations on metacognitive strategies
	Metacognitive guiding questions

TABLE 2: TYPES OF ELABORATED FEEDBACK WITH EXAMPLES (NARCISS, 2005)

Protocol of communication

The SIM in our study connects with one external module. This can be the expert knowledge module, but also the tutoring module that has a connection with the expert knowledge module and optionally the student model module.

The functionality of the connected module determines the possible functionality of the SIM. For example, if the connected module cannot give hints, the SIM cannot give hints either.

The communication from the SIM with the external modules should be done in a way that is supported by the external module. The external module can be developed by other parties; we do not want the SIM to depend on those parties to change their external module to make it connectable with the SIM. For the communication between the external module and the SIM are several standard protocols such as QTI[4], OpenMath[5], MathML[6] and OMDoc[7]. OpenMath, MathML and OMDoc are markup languages for mathematical documents (Kohlhase, 2006), QTI stands for IMS Question and Test Interoperability and is used for communication between systems and conformance testing. We want to support those parts of the protocols related to logic.

Within a protocol the messages that are send for the different entities are similar (Goguadze, 2009; Goguadze et al., 2006; Heeren and Jeuring, 2014). Heeren and Jeuring (2014) describe feedback services (see Table 3), which are requests that can be done by entities of the SIM. These or similar requests must be possible with the communication modules.

The external modules we want to communicate with are: LogAX[8] LogEx[9] and the exercise subsystem of MathBridge (Sosnovsky et al., 2012).

The architecture for the SIM should support the mentioned protocols, requests and external modules, but it should also be possible to add new or other protocols, requests and external modules.

outer loop	
– examples	predefined example exercises of a certain difficulty
– generate	makes a new exercise of a specified difficulty
inner loop	
– allfirsts	all possible next steps (based on the strategy)
– apply	application of a rewrite rule to a selected term
– diagnose	analyze a student step (details in Fig. 4)
– finished	checks whether response is accepted as an answer
– onefirst	one possible next step (based on the strategy)
– solution	worked-out solution for the current exercise
– stepsremaining	number of remaining steps (based on the strategy)
– subtasks	returns a list of subtasks of the current task
meta-information	
– exerciselist	all supported exercise classes
– rulelist	all rules in an exercise class
– rulesinfo	detailed information about rules in an exercise class
– strategyinfo	information about the strategy of an exercise class

TABLE 3: FEEDBACK SERVICES (HEEREN AND JEURING, 2014)

[4] http://www.imsglobal.org/question/index.html
[5] https://www.openmath.org/
[6] https://www.w3.org/Math/
[7] http://www.omdoc.org/
[8] http://ideas.cs.uu.nl/logax/
[9] http://ideas.cs.uu.nl/logex/

Teacher preferences

The selection of exercises that has to be solved by the student, hints and feedback are educational tools that can be controlled by the tutoring module, but in a system without a tutoring module, it may be desirable for the teacher to have control over the provision of hints and feedback (see 4.2.2). In that situation it should be possible for the teacher to control this from the SIM. This may be done by settings coded in the software or by loading some kind of configuration file that holds the settings.

Exercise answer

Answers to exercises can differ between connected modules and different types of logic exercises. Some external modules only support formulas as an answer, others need to have a combination of formulas and rules. Answers can be complete derivations but also steps of those derivations. Answers on logic exercises can be given top-down, but when proving equations a top-down combined with a bottom-up approach is also possible, and when answering an induction exercise case-structures can be used. Besides formulas and logic rules answers can also contain line numbers (used to refer to when rules are applied), and comparison signs (used with inductive logic) and also text (can be used with inductive logic and predicate logic).

Logic rule

Whether logic rules are needed depends on the connected module; not every connected module uses the logic rules as part of an answer. When logic rules are needed they can be provided by the connected module if it supports supplying those rules, otherwise the SIM should be able to supply a list or the student has to enter them as a text. If logic rules are retrieved from the external module, a transformation can be necessary between the rules that are understandable for the connected modules and those that are understandable for the student. Normally this is a task of the tutoring module but when this is not part of the ITS this transformation can be done by the SIM and it is also possible that although there is a tutoring module present in the ITS it is still wanted that a transformation is done by the SIM.

4.3. Non-functional requirements of the Student Interaction Module

To describe the non-functional requirements of the user interface of an ITS we use ISO/IEC 25010:2011 (“ISO/IEC 25010,” 2011), which is a standard to define quality standards for computer systems and software. These standards consists of two models: product quality and quality in use which are respectively divided into eight and five quality characteristics, which all are divided in several sub-characteristics (See Figure 4).

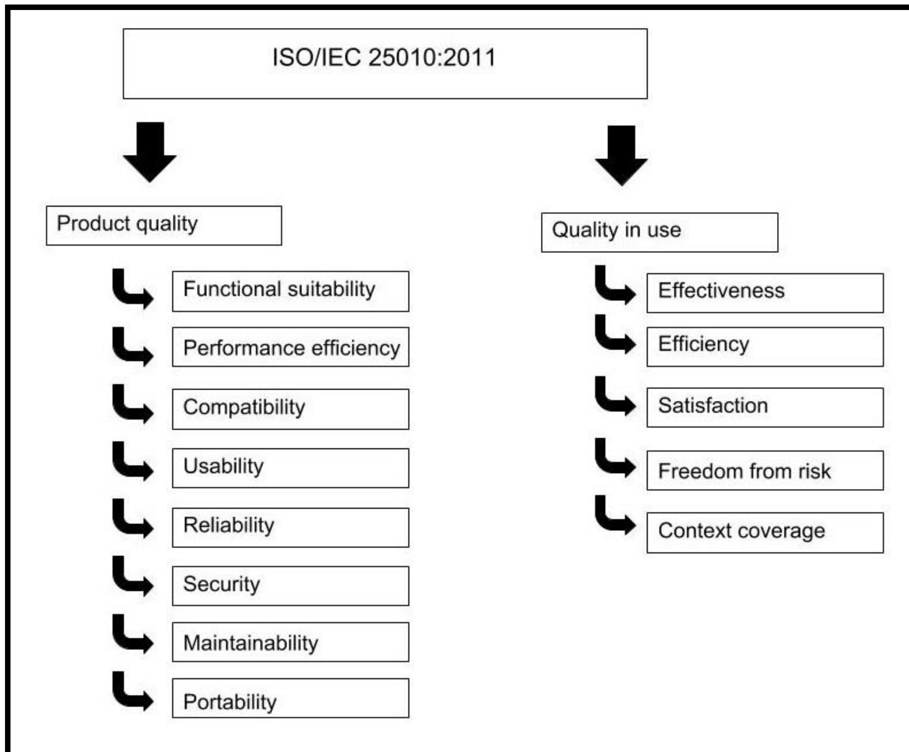


FIGURE 4: ISO/IEC 25010:201

Because the SIM architecture is a general architecture for SIMs we cannot define concrete, measurable, values for the quality characteristics, that is why we define them as quality guidelines, that should be taken into consideration during development of the architecture of the SIM and the SIM software.

Below we describe each of the quality characteristics with the concerns of the stakeholders that should be taken into consideration.

Product Quality

Functional suitability

Functional suitability is important to the students and teachers. It is important that the SIM is complete and correct and that it suits the needs of the student.

Performance efficiency

Performance efficiency is important to the students. In the study of Teeuwen (2016) is concluded that the response time of the complete system should close to 1 second than to 10 seconds.

Compatibility

Compatibility is important for the teachers, students and educational institutes. The student needs a broad set of tasks and exercises, that has to be supplied by the teachers who use an ITS to do this. The educational institutes want that this is done in a cost-effective way; they want an ITS that supports a broad set of tasks and exercises. In order to get a broad set of tasks and exercises it is important that the SIM is compatible (can connect) with many different ITSs and export modules.

Usability

Usability is important to the students and the teachers. The student needs an SIM that is easy to learn and to use. For the teacher it is important that learning settings can be changed easily.

Reliability

Reliability is important to the students and the system administrators. Students need a system that is available to do their exercises as much as possible. System administrators are responsible to keep a system available for the users. Patvarczki et al. (2009) did a study about robustness which is an important aspect of an ITS.

Security

Security is important to students, teachers and educational institutes. Students want their personal information and learning progress stored securely, teachers want to have a way to identify the student that is working on an exercise, and educational institutes have to comply with laws and regulations on the matter of security of personal data.

Security of the SIM depends on the possibilities of the tutoring module and the student model module if those are connected.

Maintainability

Maintainability is important for the developers of the SIM and the educational institutes. According to Murray (1999) the cost of development of 1 hour learning in an ITS is 300 hours. These costs can be reduced if parts of the software code can be reused. Reuse of code can be improved by creating modules with clear responsibilities. For the developers it is also important that the software can easily be modified if modules of other ITSs are added.

According to Brusilovsky (1995) the main requirements for an integration-oriented architecture are reusability and flexibility of the components. It should be possible to reuse a component of another ITS, to integrate a new component into the ITS and to replace an existing component with a similar one.

Portability

Portability is important to the system administrator. A system administrator wants software that can easily be installed on different operating systems, because he wants to be able to make applications available in a cost-effective manner, which also meets the security requirements. Therefore he wants to install the software easily on a version of an operating system that suits his needs the best at a given moment. Alpert et al. (1999) pointed out that a user interface or, in our study, an SIM should work platform independent.

Quality in use

Effectiveness

Effectiveness is important for the students who want to learn all the learning material quickly and for the teachers who want that the students to learn the learning material quickly, but this characteristic is less important for the SIM because it only presents the information created by the rest of the ITS.

Efficiency

Efficiency is less important because the SIM is only a frond-end application; storage of data, calculations and other processes that use system resources are done by the connected modules.

Satisfaction

Satisfaction is important to students. The SIM should have a highly interactive user experience (Alpert et al., 1999), but the possibilities for this also depend on the connected modules and the settings of the ITS regarding when what kind of feedback is given.

Freedom of risk

There are no risks in this quality that can be caused by the SIM.

Context coverage

Context coverage is important to students; students should be able to use all the functionality of the SIM.

5. The SIM Architecture

In the previous chapter the concerns of the stakeholders and the functional and non-functional requirements of the SIM are described. In this chapter the architectural drivers and trade-offs between the requirements are described. The architectural drivers are the most important requirements for the SIM and are used to determine the priority of the requirements when there is a possible conflict between those. The architectural drivers and trade-offs are used for the architecture, which is also described in this chapter.

The trade-offs are the answer to RQ. 2 of this study.

5.1. Architectural drivers

In the last two paragraphs we have determined the functional requirements and quality guidelines for the SIM in this paragraph we identify the most important of those.

A requirement or an quality guideline is important if it contributes to the goal of architecture of the SIM. The goal of the SIM is to cost-effectively present learning materials to students from different ITSs based on a four-model architecture. The architecture should be the basis for developing such SIMs. Other requirements are still important, but if there is a trade-off and one of the requirements of this paragraph is involved, a solution in favour of these requirements will be chosen.

Cost-effectiveness can be achieved by low development and maintenance cost of an SIM, by using software of low complexity and the reuse of software components, and by connecting the SIM to more different modules with different learning materials. Low complexity and reuse of software components are part of the maintainability characteristic and connecting the SIM to more different modules with different learning material is part of the compatibility characteristic.

The functional requirements and especially the optionality of functionality is important for the SIM. The SIM should connect with different modules of ITSs, those modules can have different functionality which the SIM should be able to support that functionality.

5.2. Trade-offs

In this paragraph the trade-offs are described. This is the answer to RQ. 2 of our study. The trade-offs are divided into trade-offs between the requirements and trade-offs related to the architectural design of the SIM.

5.2.1. Trade-offs between the requirements

The compatibility and maintainability requirements are conflicting with each other. Compatibility for the SIM means that it can connect to modules of other ITSs and support the functionality of those modules.

Brooks and Kugler identify complexity as one of the difficulties with maintaining software (Brooks and Kugler, 1987), but the compatibility of the SIM can lead to complexity because of the next three reasons:

Firstly the communication between modules is not clearly defined. There are some communication standards but also other communication protocols can be used and it is likely that the same kind of modules from different ITSs communicate differently with the SIM. This means that for almost every module that can be connected separate functions for communication have to be build.

Secondly the learning material (exercises, feedback and hints) that is offered by the connected modules is not always the same. The modules offer different forms of logic, where the answers can be formulas, rules or a combination of both. Answers can be given top-down but it is also possible that a bottom-up answering is allowed as well. Feedback is

optional and can be given every step or only after the complete answer. Hints are also optional, and can be given in various ways like a sentence with a hint, showing the next step or showing the complete derivation. All these possibilities can lead to complex software.

Thirdly the optionality of the tutoring and the student model modules in the complete ITS can cause complexity for the SIM. The tutoring model or a combination of the tutoring and the student model can control the exercises that are presented to the student and this can influence how the exercises are presented and whether or not hints and feedback is shown and in which form these are shown. When both models do not exist in the ITS it can be needed that the SIM provides part of this functionality.

Besides the conflict between compatibility and maintainability there is also a conflict between the optionality of a parts of the functionality and the maintainability that is also related to the complexity.

Part of the requirements is that hints and feedback can be turned on and off and the selection of exercises can be done by the student but can also be done by the teacher or by a connected module. In addition, not every combination of connected models offers the same functionality, which also leads to optional functionality. Optional functionality can lead to complex software because every possibility can have consequences for the working of the entire software. These consequences can be limited by using a good architecture and software design.

In addition, once the SIM is build it should be possible to add more connections to other ITSs in order to provide the students with, for example, more exercises or different exercises or other student related support from another tutoring model in combination with a student model. This can lead to changes in the existing SIM. Lehman's second law points out that changes in software can increase the complexity of it (Lehman, 1980).

5.2.2. Architecture design trade-offs

In this paragraph we discuss some high-level architecture design trade-offs that form the basis of the architecture of the SIM. We look the following questions:

- Web based or not web based?
- Plug-in architecture or product line architecture?
- Which application architecture?

For each of these items we look at the product quality characteristics of the non-functional requirements for the architecture of the SIM these are the most relevant characteristics (see 0).

5.2.2.1. Web based or not web based

We want to investigate an architecture for an SIM that can be connected to modules of several ITSs and LEs and that can be used by many students. The maximum number of students using it at the same time should rather be limited by the capacity of the connected modules than by the capacity of the SIM itself. Table 4 presents a comparison for the product quality characteristics for a web based and a not web based architecture.

Web based or not web based does not affect the functional suitability; on both it is possible to make an application that suits the needs of the students.

The performance efficiency of the SIM mainly depends on the number of students using it. The target audience for the sim is large, but the SIM is only a frond-end that can be installed on the device of the student or can be retrieved from a webserver; so the CPU capacity needed for running the SIM for the students will be divided over the devices used by those students. A possible bottleneck can be the performance connection to the external modules. Ritter and Koedinger (1996) point to the possible delay in the connection when using an

internet connection, but this can also be a problem on an internal network of the organisation. The biggest problem can be the performance of the connected external modules when many students use a connected external module, but this possible problem is the same for a web based and a not web based architecture.

The compatibility is the same for both architectures; a web based and a not web based application can both connect to the same external modules.

The usability of both options can be the same if both can be used from a device of the student; when it can be used from a device of the student, the student can work time and place independent. If the SIM only can be used on a device of the organisation this will be limited. Although it is likely that a web based architecture offers the possibility to be used on a device of the student, it is still possible that this is limited to specific devices.

The reliability of both options can be the same.

Security is not a real issue for the SIM; it mainly depends on the security measures of the connected modules (see 0), but if the SIM can be used outside the network of the organisation it can make a point of access for hacking. The security risks of this do not depend on a web based or not a web based architecture.

The maintainability of both options can be the same.

The portability of the web based architecture is better because this can be used within a browser installed on the device of the student, whereas the not web based architecture can only be used if can be used on specific the operating systems or the SIM has to be developed in such a way that it can run on all operating systems, but that would affect the maintainability of the SIM.

Alpert et.al. (1999) mention benefits of using the web such as: capable to handle a great number of students at the same time and the students can work place and time independent. A web-based architecture also uses infrastructure that is available on every computer, laptop, tablet and mobile phone, which gives a better portability for the SIM as a not web based solution would do.

Because the SIM only interacts with the student and communicates with the connected modules and there are no processes that need much CPU from the computer, we think that an architecture with a single webpage, hosted on one of more webservers (scalability) will be enough. This webpage runs within a browser on a device of the student and directly connects with the connected module on a central place. With this option the number of users at the same time is limited to the boundaries of the connected modules.

Architecture → **Product Quality ↓**	**Web based**	**Not web based**
Functional suitability	+	+
Performance efficiency	+	+
Compatibility	+	+
Usability	+?	+?
Reliability	+	+
Security	+?	+?
Maintainability	+	+
Portability	+	+/-

+ = optimal
+/- = sufficient
? = has attention points

TABLE 4: COMPARISON PRODUCT QUALITY CHARACTERISTICS FOR A WEB BASED AND A NOT WEB BASED ARCHITECTURE

5.2.2.2. Plug-in or product line architecture

From the functional requirements we know that the SIM has several optional functionalities that depend on the possibilities of the connected module and on the preferences of the teacher at a certain moment. Therefore we want to use an architecture that can handle variability we look at the plug-in architecture and the product line architecture.

The plug-in architecture is characterized by a host application with base functionality to which plug-ins, that also can be developed by other parties, with variable functionality can be connected at runtime (Birsan, 2005). Because of this plug-in systems need clearly defined interfaces that can add functionality to the system (Birsan, 2005). Example of plug-in architectures are: the development tool Eclipse and the web browser Chrome with its extensions.

The Product line architecture is an architecture of similar products customized for a particular market segment or for a particular purpose, where the variable functionality is part of a product (Northrop et al., 2007). An example of a product line is Windows with its home, pro, enterprise etc. editions.

Table 5 presents a comparison of the product quality characteristics of ISO 25010 for the plug-in and the product line architecture. The scores are based on the theory of product line architectures in Nothrop (2007), Dermeval et al. (2017) and Marcolino and Barbosa (2017) and the plug-in architecture of Birsan (2005). All these architectures describe a complete ITS and not only the part that communicates with the student.

Performance efficiency is by both sufficient but because in both architectures the interface and the base functionality have to be more general, so that several third party and variable components can be used.

When it comes to reliability the biggest concern is the reliability of the connected modules. We want to be able to connect to modules developed and maintained by others, thus for the reliability of the SIM we depend on those other parties. This does not affect the comparison between a plug-in or a product line architecture, because both architectures have this problem.

When it comes to security there is a risk with the plug-in architecture when third parties connect plug-ins; you do not have control over those plug-ins, which makes it possible to send data from the plug-in elsewhere.

When it comes to maintainability with a plug-in architecture it is possible that the cause of defects is not always clear; is it a defect in the system or is it a defect in a plug-in? A plug-in architecture can also make it more difficult to change the system, especially when this affects the interface or internal functions used by plug-ins.

If we look at the issues in the comparison we see that the disadvantages of the plug-in architecture are related to plug-ins created by other parties, but when we develop those plug-ins ourselves the disadvantages would not exist.

The advantage of a plug-in architecture is that other parties can develop plug-ins. This could lead to more modules that can be connected to the SIM, which can result in a bigger variance in learning material that can be studied with it. The advantage of the product line architecture is that we can develop different products for different target groups of students. The core of the products will always be the same, but the special features for a product are only used by that product, which results in fewer lines of code and less complex software. Less complex software improves the maintainability (Brooks and Kugler, 1987).

For the SIM architecture we choose a product line architecture because we want to have low complexity since this improves the maintainability. This choice can reduce the

compatibility of the SIM, because it is possible that we cannot communicate with every external module. This can be solved by other parties by creating a connector between the external module and the SIM, which converts messages from the external module into messages that can be used by the SIM and vice versa.

Architecture → Product Quality ↓	Plug-in	Product line
Functional suitability	+	+
Performance efficiency	+/-	+/-
Compatibility	+	+
Usability	+	+
Reliability	+?	+?
Security	+?	+?
Maintainability	+?	+
Portability	+	+

+ = optimal
+/- = sufficient
? = has attention points

TABLE 5: COMPARISON PRODUCT QUALITY CHARACTERISTICS FOR PLUG-IN AND PRODUCT LINE ARCHITECTURE FOR THE SIM

5.2.2.3. Application architecture

For the application architecture we take three options into consideration: Model-View-Control (MVC), Blackboard and Flux. There are more options when it comes to architectures for a web-only client but we think that many other architectures have much in common with these ones.

The MVC architecture (see Figure 5) has a model in which all the data is stored, a view for everything that has to presented to the user for which it queries the model when data is needed and a control for handling user input, updating values in the model and triggering the update of the view (Krasner and Pope, 1998).

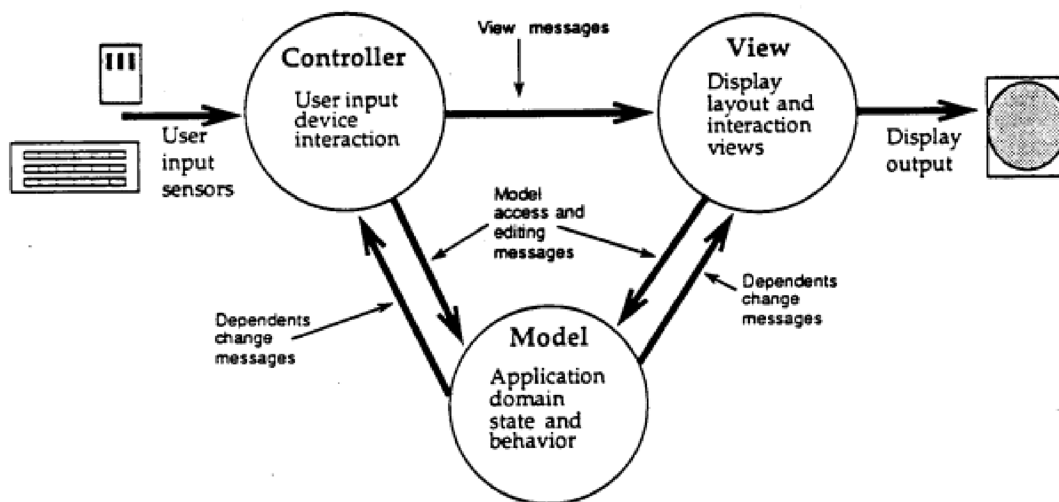


FIGURE 5: MODEL-VIEW-CONTROL (KRASNER AND POPE, 1998)

The Blackboard architecture (see Figure 6) has a part called "blackboard" for the user interaction (input and output), knowledge sources and a control component. The knowledge sources are independent modules needed for the solving of problems that know when they can contribute to the solution. The control module controls, based on the knowledge sources, the actions for the complete system (Corkill, 1991).

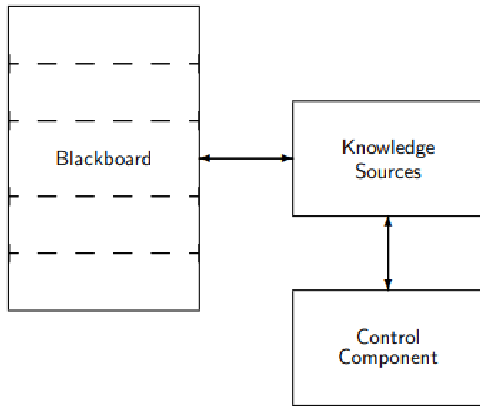


FIGURE 6: BLACKBOARD (CORKILL, 1991)

The Flux architecture has a dispatcher that handles all actions and send those to the store where the data is stored, the stores send the data to the views, and the views can create new actions and there can be actions connected to the dispatcher, these are helper methods which the dispatcher can use (Boduch, 2016; "Flux," 2019).

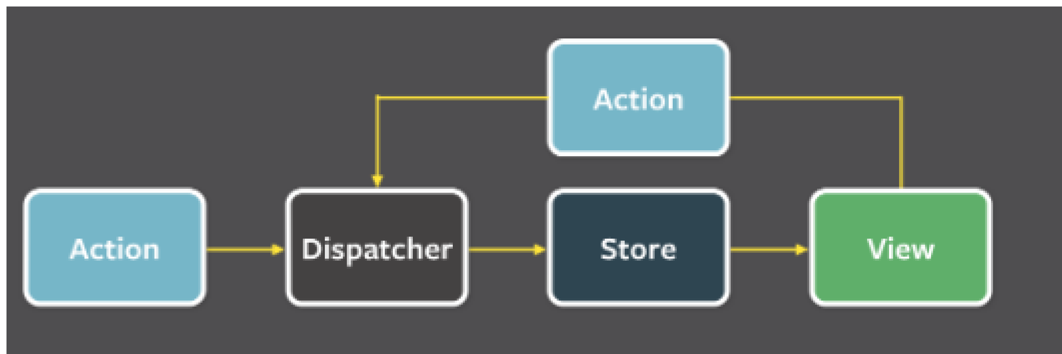


FIGURE 7: FLUX ("FLUX," 2019)

Blackboard compared to MVC and Flux is more complex because of the AI knowledge sources in it. Blackboard is more an architecture to be used with complex systems (Corkill, 1991).

Compared to MVC has Flux a clear view on the flow within the software; the dispatcher communicates only with one or more stores, stores only communicate with views and views can have actions, that are send to the dispatcher (Boduch, 2016) and in MVC the controller updates the model and triggers the view and the view queries the model for data (Krasner and Pope, 1998). When doing complex cascading updates Flux can be more easy to understand (Boduch, 2016).

When we compare MVC with Flux (see: Table 6) we expect both to score optimal on all product quality characteristics of ISO25010 for the SIM. This expectation is based on that we did not find any articles or blogs on internet about attention points, except for attention points mentioned earlier. The possible complexity with cascading updates in MVC can affect the maintainability, but we do not expect cascading updates in our SIM architecture.

For the SIM we choose to use an MVC architecture, because we are more familiar with this kind of architecture.

Architecture → **Product Quality ↓**	MVC	Flux
Functional suitability	+	+
Performance efficiency	+	+
Compatibility	+	+
Usability	+	+
Reliability	+	+
Security	+	+
Maintainability	+	+
Portability	+	+

+ = optimal
+/- = sufficient
? = has attention points

TABLE 6: COMPARISON PRODUCT QUALITY CHARACTERISTICS MVC AND FLUX BASED ARCHITECTURE

5.3. Architecture views

In this chapter we look at the architecture of the SIM. The architecture consists of: the entity view describing the main entities of the SIM using a simplified entity relation diagram (paragraph 5.3.1), the overall view describing the main flow of the SIM (paragraph 5.3.2), the functional view describing the possible functionalities of the SIM using a simplified entity model (paragraph 5.3.3), and the development view describing how the SIM is divided in technical components and functions within those components (paragraph 5.3.4).

5.3.1. Entity view

In this paragraph the entity view is described, this gives a picture of the most important entities and the relationships between them.

Based on the functional requirements from paragraph 4.2 we have defined the following entities: exercises, answers, rules, hints and feedback (see Figure 8 for the entities and the relations between them.) We have chosen these entities based on the principle of high cohesion within the entities and low coupling between the entities (Ingeno, 2018), which means that everything within an entity should have a strong relation, but the entities have a low relation with another. Below we give a description of each of the entities.

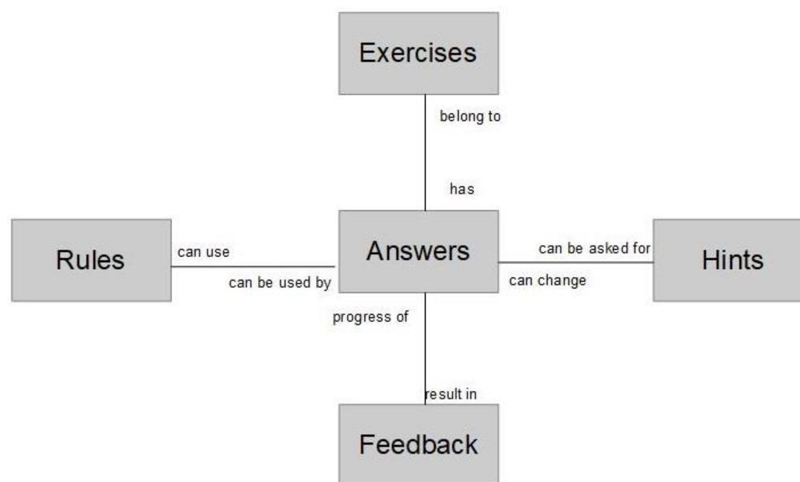


FIGURE 8: ENTITIES OF THE SIM

Exercises

Exercises are the logic issues that have to be solved by the student. These can be pre-defined exercises in the connected modules or randomly generated exercises by the connected modules or exercises defined by the student. The selection of the exercise can be done by the module or by the SIM or by the student, depending on the possibilities of the connected module and the preferences of the teacher.

Exercises can belong to groups of exercises based on education goal or difficulty of the exercises. This is in our architecture seen as a property of an exercise, but it is possible that these groups of exercises may be seen as a separate entity in some situations.

Answers

Answers are the replies given by the students to exercises. The format of an answer depends on the connected external modules of the SIM. Answers can contain formulas, applied rules, comparison signs (from induction logic) and line numbers. Answers can be given step-based or a complete answer on the complete exercise. A step of an answer can be a single line in the solution of the exercise, or a part of such an answer or a part of a case when solving an induction exercise. Steps of an answer or the complete answer can be send to the connected module depending on the possibilities of the connected module and the preferences of the teacher (see 4.2.2).

Rules

Rules can be used when answering exercises. Rules can be provided by the connected module or be determined by the SIM . Rules are only needed if the connected module needs rules as a part of an answer.

Although rules are used by answer and can depend on the current status of the answer, we have decided to make a separate entity for rules. This way answer will not be responsible for retrieving the rules from the external module or for determining what rules may be used with a step of the answer.

Hints

Hints can be available for the student during answering the exercises. Hints can be a line of text, an example, the complete derivation, the next step in the solution or completing the solution of the student. The availability of hints and the kind of hints depend on the connected module and on the preferences of the teacher.

Although hints are closely related to answers; they use the current status of the answer and can have an effect on the answer when a next step or completion of the answer is requested, we have decided to make a separate entity for hints.

Feedback

Feedback can have a KR part; depending on the result of the diagnose done by the external module the next action for the SIM is given by the external module or determined by the SIM. If no KR feedback is given the SIM has to determine the next action based on the input of the student (a next step or a new exercise).

Feedback is given by the connected module after an answer or an answer-step is checked by the connected module. Feedback can be only an indicator for correct or incorrect but can also be an explanation, or a hint, or an update of the overall results of the student, or a combination of these items.

The kind of feedback that available for the student depends on the connected module and on the preferences of the teacher.

5.3.2. Overall view

Figure 9 presents a functional overall view of the SIM. The program starts with the Exercise module where an exercise is selected. The selection of an exercise can be done by the connected external module or by the SIM. Other options are that the student selects the exercise or the student defines an exercise.

When the exercise is selected, the student can enter the answer in the Answer module. This is done stepwise. After each step the step is send to the external module, which checks the step. If the student thinks the answer is complete the complete answer is send to the external module to check the complete answer.

If supported the student can ask for hints when answering exercises. What kind of hints are available depends on the connected module and the settings in the SIM. Some hints can change the current status of the answer, for example when asked for a next step as hint, this step is added to the already given steps. Other hints are only text and do not change the answer.

During answering rules can be used. Rules are provided by the Rules module and can be, if available, retrieved from the connected external module: otherwise the Rules module itself provides the rules. The possible rules are returned to the Answer module, where the rules can be used.

The answers on steps and complete exercises are sent to the external module that will check the step or answer. The external module will return feedback and that is received by the Feedback module. The Feedback module will display feedback messages if those are supported by the external module and the teacher has not disabled it. Next the Feedback module determines the next step to take: back to the Answer module and redo the last step because it was not correct, back to the Answer module for a next step because the exercise is not completed or back to the Exercise module because the answer was completed.

The exercise module will select a new exercise or request the external module for a new exercise. The process will end when there are no more exercises or when the student ends the program.

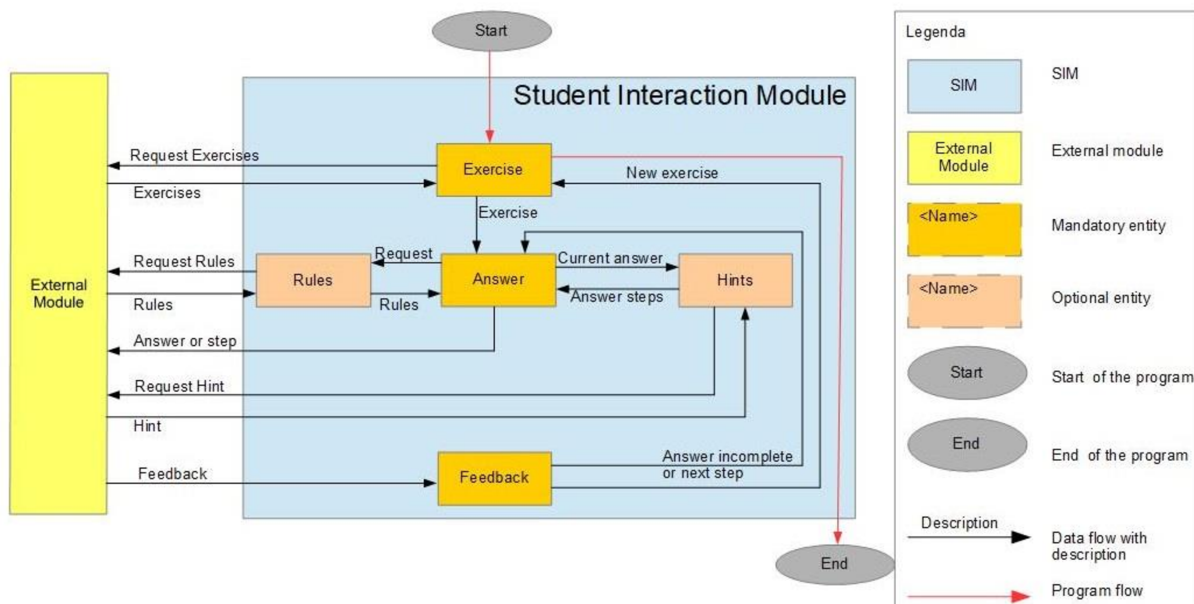


FIGURE 9: FUNCTIONAL OVERALL VIEW SIM

5.3.3. Functional view

In this paragraph we describe the functional view using a simplified feature model for the SIM. This model gives a view on the functional options of the SIM. Normally this kind of model can also be used to define the relationships and exclusions between features; we have omitted this from our scheme and description; with this architecture we want to describe the possible features of SIMs in general. The relationships and exclusions are part of a concrete modules developed for SIMs with this architecture.

This functional view contains many implementation aspects, because it was initially thought to only use this view. In retrospect, this turned out to be insufficiently clear and a development view was added.

Whether features exist in a developed SIM depend on the possibilities of the connected module, the connected module must support those features. Features can, within the possibilities and limitations of the connected module, also depend on the preferences of the teacher; a teacher can decide how exercises are selected for the students, how rules are shown, whether answers have to be given in steps or not, which parts an answer consists of and whether hints and feedback are shown. The preferences of the teacher can be part of the developed modules but can also be settings of the SIM, in which case the teacher preferences are an extra supporting entity that can be used by all other entities. To reduce the complexity we have omitted this from our schema.

This view on the architecture based on the MVC-model; our first thought was to make only one view on the architecture and combine the features and the implementation aspects in this view. Later on we decided to add the development view, but we left the feature model as it was.

Every paragraph starts with a schema followed by an explanation of the schema. This explanation consists of details and can be skipped completely or partly depending on the interest of the reader.

Below you will find the legend that can be used by every scheme.

If a feature is not available in every product-line, we have made it **optional**, all other features are **mandatory**.

Abstract is a feature when we expect it will not be mapped to any implementation artefact, all other features are **concrete** (Thüm et al., 2011). When programming a version of the SIM it is still possible that abstract features will become part of the software, because it is needed by the programming language or needed because of design aspects. For example. it makes the code better understandable and maintainable.

In the diagrams the features are connected with lines, the features below a feature are the child-features of that feature. Some diagrams have an **or group** to indicate an exclusive or, that means that only one of the options can be chosen in a product of the product line.

5.3.3.1. Design decisions

In this paragraph some overall design decisions for the feature model of the SIM are mentioned.

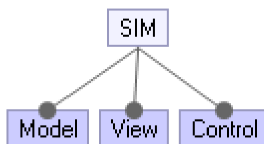
Communication

All entities may need to communicate with the external module. Communication is preparing messages to be send, sending those messages, receiving return messages and decoding those messages, where all these actions use a certain protocol and communicate with a configured ip-address. Although every entity has its own messages we think that there are many similarities between the actions related to communication that has to be done. This is why we have designed these dedicated features for the communication with external modules: CommunicationModel, CommunicationControl and CommunicationView.

Exercises

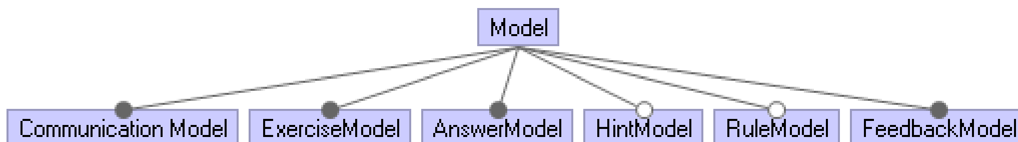
Exercises have to be selected and showed to the student. We have made showing of exercises part of answer, because with a step-based answer every step can be seen as a new exercise that has to be answered. The task of exercise in our models is the selection of new exercises.

5.3.3.2. SIM



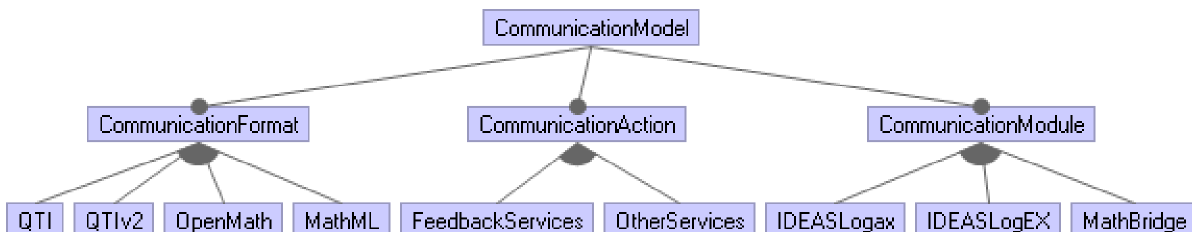
Sim consists of Model, View and Control (the basic parts of an MVC-model architecture). Model is responsible for all definitions of elements used in the application and it is responsible for the communication with other modules. View is responsible for the presentation of everything on the device of the student. Control is responsible for all changes in values of elements in Model. These changes are caused by input in the View and new values received from other modules in Model.

5.3.3.3. Model



Model consists of CommunicationModel, ExerciseModel, AnswerModel, FeedbackModel and optional HintModel when hints are available and RuleModel, when rules are used.

5.3.3.3.1. CommunicationModel



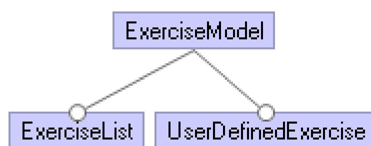
CommunicationModel contains the definitions of what in which way should be communicated with which module. It consists of CommunicationFormat, CommunicationAction and CommunicationModule.

CommunicationFormat is responsible for the definitions of the possible communication formats. These communication formats can in our model be QTI or QTIv2 or OpenMath or MathML, but other formats should also be possible to add.

CommunicationAction is responsible for the definition of all possible communication actions. The available actions depend on the possibilities of the attached module. In the theory we have found the feedback services from Heeren and Jeuring (2014) (see Table 3) those are handled by FeedbackServices, but we expect there will be other similar actions in other systems, those are handled by OtherServices.

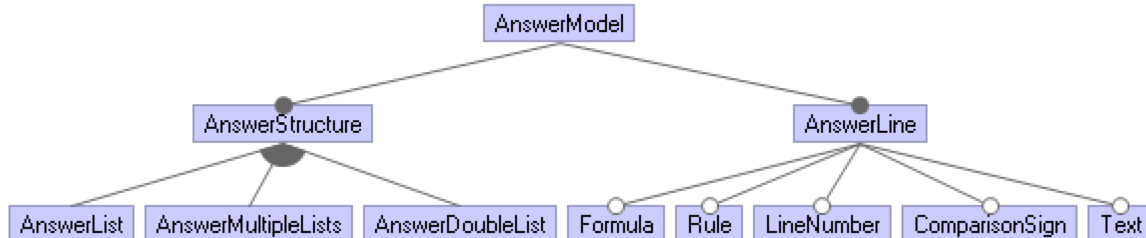
CommunicationModule is responsible for the definition of how the communication should be done with which module. This describes the module's URL, the necessary credentials, the communication format it uses and the type of communication action it uses. In our model IDEASLogax, IDEASLogEX and Mathbridge are mentioned, but also other ITSs or modules from ITSs should be possible to add.

5.3.3.3.2. ExerciseModel



ExerciseModel is responsible for the definition of ExerciseList and for storing the values of a list of exercises received from a connected module and for storing the user-defined exercises. The features in ExerciseModel are optional and only needed if the user or the SIM can select an exercise (ExerciseList) or if the user can define an exercise (UserDefinedExercise).

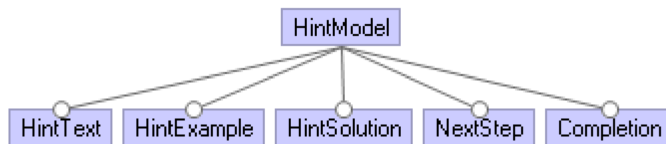
5.3.3.3.3. AnswerModel



AnswerModel is responsible for the definition of an answer and for storing given answers. The definition of an answer consists of two parts: the structure of an answer (AnswerStructure) and the content of a line within a used answer structure (AnswerLine). Both depend on the answer format expected by the external module. The storing of the given answers is done within the variables defined by this structure.

An answer structure can be a list of answer lines (AnswerList), or a list with multiple lists of answer lines (case structure) (AnswerMultipleLists) or a list in which answer lines can be added top-down and bottom-up (AnswerDoubleList). An answer line can be a line number, a rule, a formula, a comparison sign (inductive logic) or text or any combination of those.

5.3.3.3.4. HintModel

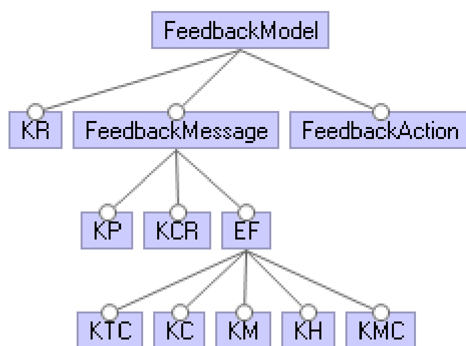


HintModel is responsible for the definition of hints and storing received hints. Hints are divided in: textual hints (HintText), example solutions (HintExample), the solution of the exercise (HintSolution), the next step in solving the exercise (NextStep) and completion of the entered solution (Completion). Availability of each of these options depends on the possibilities of the connected module.

5.3.3.3.5. RuleModel

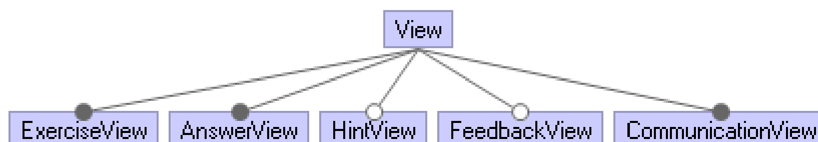
The RuleModel is responsible for storing the logic rules that can be used by answering exercises. Rulemodel is optional, while it is only needed if logic rules are part of the answer. RuleModel has no child-features.

5.3.3.3.6. FeedbackModel



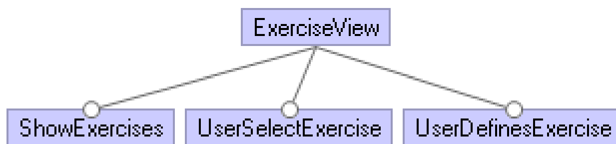
FeedbackModel is responsible for storing the feedback returned by the connected external module. This can be only a result (KR-feedback) but also a combination of a result and messages of other types of feedback (KP and KCR feedback and the EF feedback messages) or it can be only an action that has to be done next (FeedbackAction move to next step or go to the next exercise). Availability of the messages and actions depend on the possibilities of the connected external module and the preferences of the teacher, but there should be always one indication for the SIM to determine the next step in the program (go to the same step in the answer, go to the next step in the answer, go to the next exercise).

5.3.3.4. View



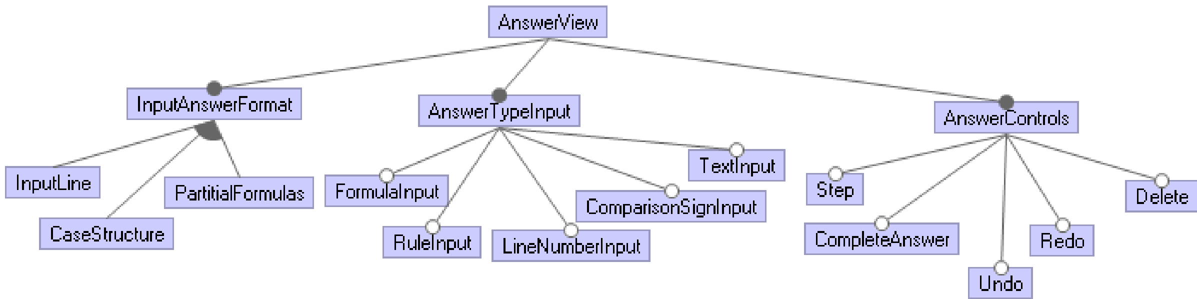
View consists of ExerciseView, AnswerView, FeedbackView, CommunicationView and optional HintView when hints are available.

5.3.3.4.1. ExerciseView



ExerciseView is responsible for the presentation of available exercises (ShowExercises) that can be selected by the student (UserSelectExercise) and for the presentation of an option in which the student can define an exercise.

5.3.3.4.2. AnswerView

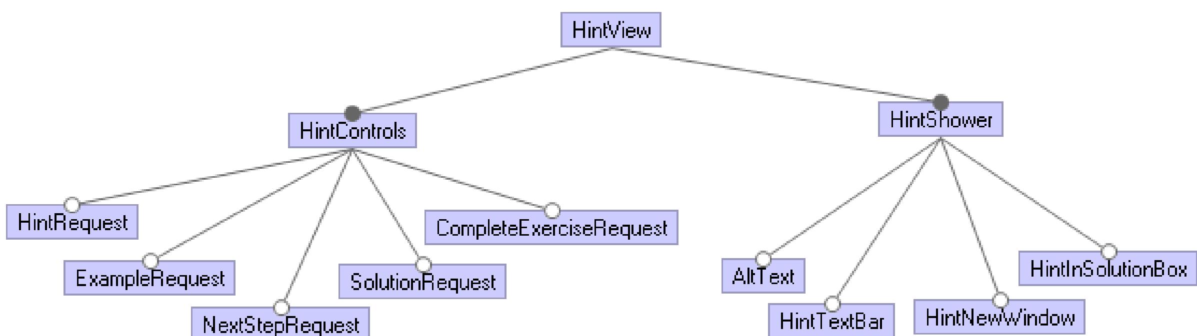


AnswerView is responsible for presenting the part in which the student answers exercises. AnswerView consists of the way the input of an answer is presented (InputAnswerFormat), one or more answer types that together form an answer (AnswerTypeInput) and controls for changing the status of an answer or changing the answer (AnswerControls).

The format of the input of an answer (InputAnswerFormat) can be an input line (InputLine) or a case structure with input lines (CaseStructure) or a block in which the answer is given in parts, depending on the used rule (PartialFormulas). The way the input for an answer is presented (AnswerTypeInput) can contain multiple items from: a rule that is used, a formula, one or more line numbers and a comparison sign.

The controls for an answer (AnswerControls) are: Step to indicate that the answer is for one step, CompleteAnswer to indicate that the exercise is completed, Undo to delete the last step, Redo to undo the deletion of the last action and Delete to delete all steps from the point the delete button is pressed.

5.3.3.4.3. HintView



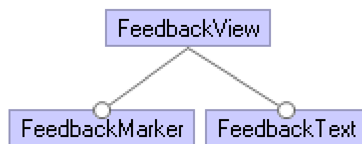
HintView is responsible for showing the options to request hints and showing hints after they have been received from the connected module and stored in HintModel. HintView consists of HintControls, for showing the controls with which the student can ask for hints and HintShower for showing the hints.

HintControls have several optional options HintRequest for asking a hint, ExampleRequest to ask for an example solution, NextStepRequest to ask to give the next step in the solution,

SolutionRequest to ask for the complete solution of the exercise, CompleteExerciseRequest to ask for the completion of the answer, but depending on the connected module other options are possible too.

HintShower offers the following options to show the hints: AltText to show the hint in a pop-up balloon above the answer, HintTextBar to show the hint in a line in the answering window, HintNewWindow to show the hint in a new window, HintInSolutionBox to show a hint in the part of the window where partial answers are given. This should not be seen as a limited set; it should be possible to add other options.

5.3.3.4.4. FeedbackView

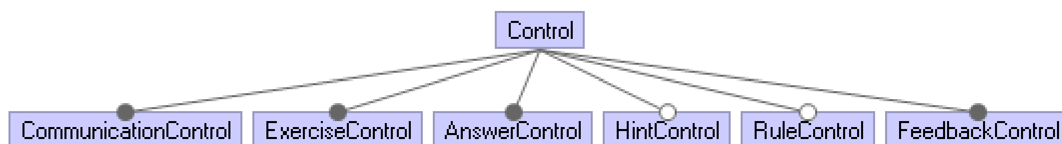


FeedbackView is responsible for the presentation of feedback results. Available options are a marker or a text or a combination of both. Feedback is only shown if the teacher has allowed this. FeedbackMarker depends on KR feedback (correct / incorrect), FeedbackText can show all kinds of feedback text messages.

5.3.3.4.5. CommunicationView

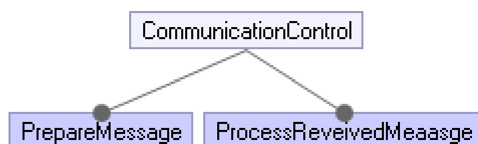
CommunicationView is responsible for showing error messages occurred during communication with the connected module and that has to be shown to the student (i.e. message that the connected module is not available). CommunicationView has no child-features.

5.3.3.5. Control



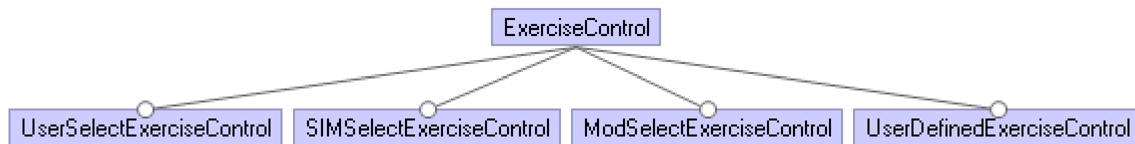
Control consists of CommunicationControl, ExerciseControl, AnswerControl, FeedbackControl and optional HintControl if hints are available in the connected module and allowed by the teacher and RuleControl if rules are used with answering the exercises.

5.3.3.5.1. CommunicationControl



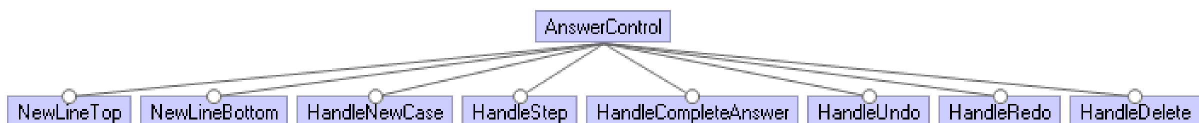
CommunicationControl is responsible for preparing messages for communication with a module and for handling messages received from a module. It consists of PrepareMessage and ProcessReceivedMessage.

5.3.3.5.2. ExerciseControl



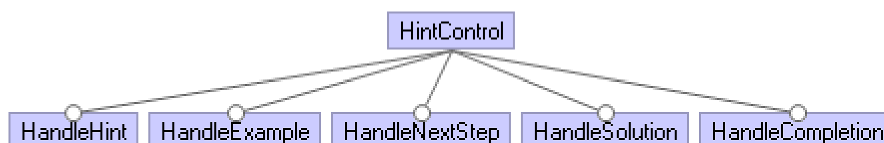
ExerciseControl is responsible for the control functions related to the selection of an exercise. A task can be selected by a student (UserSelectExerciseControl), by the SIM (SIMSelectsExerciseControl), by the connected Modules (ModSelectExerciseControle) or it can be defined by the student (UserDefinedExerciseControl) and combinations of these options are also possible, only combinations with SIMSelectsExerciseControl and ModSelectExerciseControle, because both are automatic selections and this has to be done or by the SIM or by the connected module.

5.3.3.5.3. AnswerControl



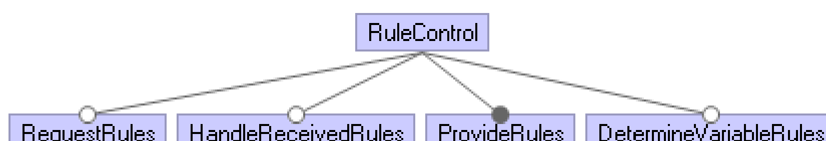
Answer control is responsible for handling all actions in AnswerView. NewLineTop is used to add a new input line at the top part of an answer. NewLineBottom is used to add a new input line at the bottom part of an answer. HandleNewCase is used to add an new case with answer lines to the answer. HandleStep is used to indicate that a step of the answer can be send by the communication model (CommunicationModel) to the attached module. HandleCompleteAnswer is used to indicate that the complete answer can be sent by the communication model (CommunicationModel) to the attached module. HandleUndo is used to undo the last entered step. HandleRedo is used to redo the last entered step. HandleDelete is used to delete all steps starting with the selected answering line, the direction of deleting depends on the direction of the answer.

5.3.3.5.4. HintControl



HintControl is responsible for handling all actions related to hints. These actions are the result of requests by the student in HintModel. These requests are: a request for a hint (HandleHint), a request for an example solution (HandleExample), a request for the next step of the solution (HandleNextStep), a request for the complete solution (HandleSolution) and a request for the completion of the answer (HandleCompletion).

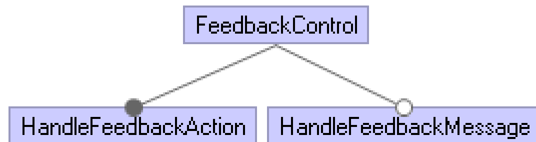
5.3.3.5.5. RuleControl



RuleControl is responsible for providing the SIM with rules (ProvideRules). These rules can be retrieved from the connected module via the communication modules (RequestRules and HandleReceeivedRules) or be queried from the values in RuleModel and RuleControl can, if needed, transform the rules (DetermineVariableRules). When rules are used in the SIM, ProvideRules is mandatory, RequestRules and HandleReceivedRules are both needed if the

rules need to be retrieved from an external module. DetermineVariableRules is only needed if rules has to be transformed for a student or for parts of the exercise.

5.3.3.5.6. FeedbackControl



FeedbackControl is responsible for handling feedback returned by the connected module after a student has given a step in the answer or a complete answer to an exercise. Feedback actions are handled by HandleFeedbackAction and feedback messages are handled by HanldeFeedbackMessage. Feedback of types AUC and MTF are also handled by HandleFeedbackAction they react on KR feedback given by the connected module. HanldeFeedbackMessage can also be used to transform feedback messages from codes used by the connected module into understandable messages for the student.

5.3.4. Development view

In the feature model we have shown options for features of the SIM. The entities hints and rules are optional, and all the entities have several optional features, depending on the SIM that is needed for a specific external module or a combination of external modules. In this paragraph we present another view: the development view to show how the features can be modelled in modules developed. We use a product line architecture for the modules, which means that from every module there can be more than one version of that module. The SIM for a specific external module or combination of modules can be built by combining those reusable modules.

In the next paragraphs we describe the development view using several diagrams. In the overall view we describe the complete SIM with entities and the main data streams between them.

In the other diagrams we show how the entities are divided into modules, the functions and components of these modules, the data flows and triggers between the modules, functions and components and the student and modules responsible for the input and output of an entity on the top and bottom side of each diagram. To keep our diagrams clear, we only show the most important items.

In the overall and the entity diagrams the symbols from Figure 10 are used.

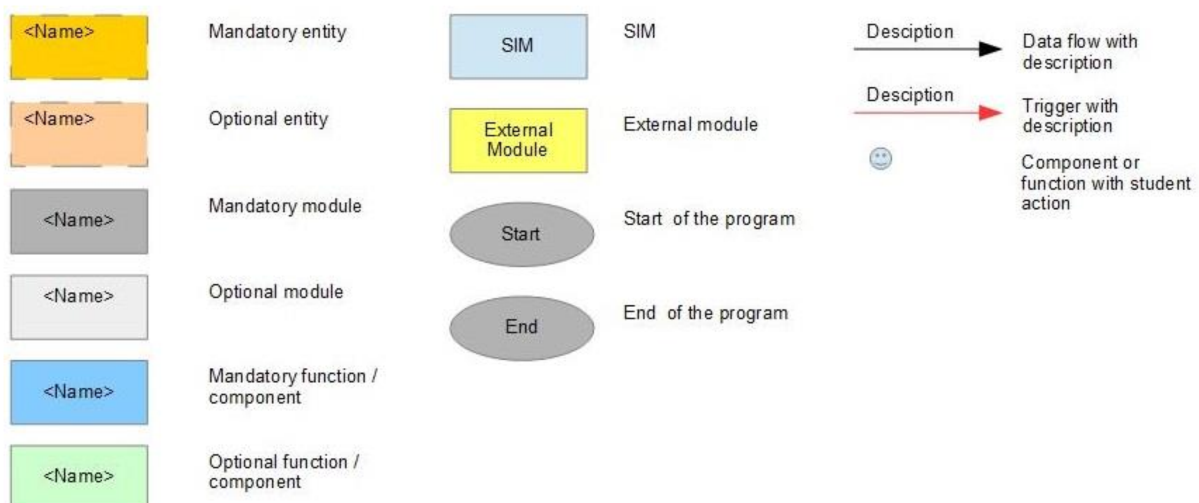


FIGURE 10: USED SYMBOLS

5.3.4.1. Design decision: Program

To control the specific versions of modules and the optionality of modules we use an extra entity: Program that consists of ProgramControl, ProgramModel and ProgramView. Program is responsible for controlling the flow of the SIM and therefore it needs to know the used modules within a specific product of our product line for the SIM. By using Program, modules do not need to have knowledge about the other connected modules. This makes modules independent from each other and there is not a need for special modules for combinations of models.

Program can also be used to reduce the number of versions of a component. When we look at the relation between Answer and the optional component Hints, there can be several versions of answer and several versions of hints and there can be a product in the product line without hints. When Hints is available, Answer always has to send the current status of the answer to Hints, but when Hints is not available, Answer does not have to send the current status of the answer. This would double the number of versions of Answer (for every version of Answer there has to be one with and one without sending the current answer to hints). By using Program it is possible to make versions of Answer that always sent the current status of answer and let Program decide what to do with it (ignore when Hints is not available or sent to Hint if Hint is available).

5.3.4.2. Overall view

Below you find the overall view of the SIM (see Figure 11), with the entities from paragraph **Fout! Verwijzingsbron niet gevonden.** and the entity Communication from paragraph 5.3.3.1 and the entity Program from paragraph 5.3.4.1. The overall view gives a global overview of the entities and the communication between them and the communication with the connected module.

Each of the entities have a Model, a Control and a View component (see Model-View-Control Architecture in paragraph 5.2.2.3) except for Rules, which does not have a view, because it only supplies rules to Answer, that are presented in Answer; Rules does not present any information to the student.

In the next paragraphs we will describe the responsibilities of the components and the important differences with the features of each of the entities.

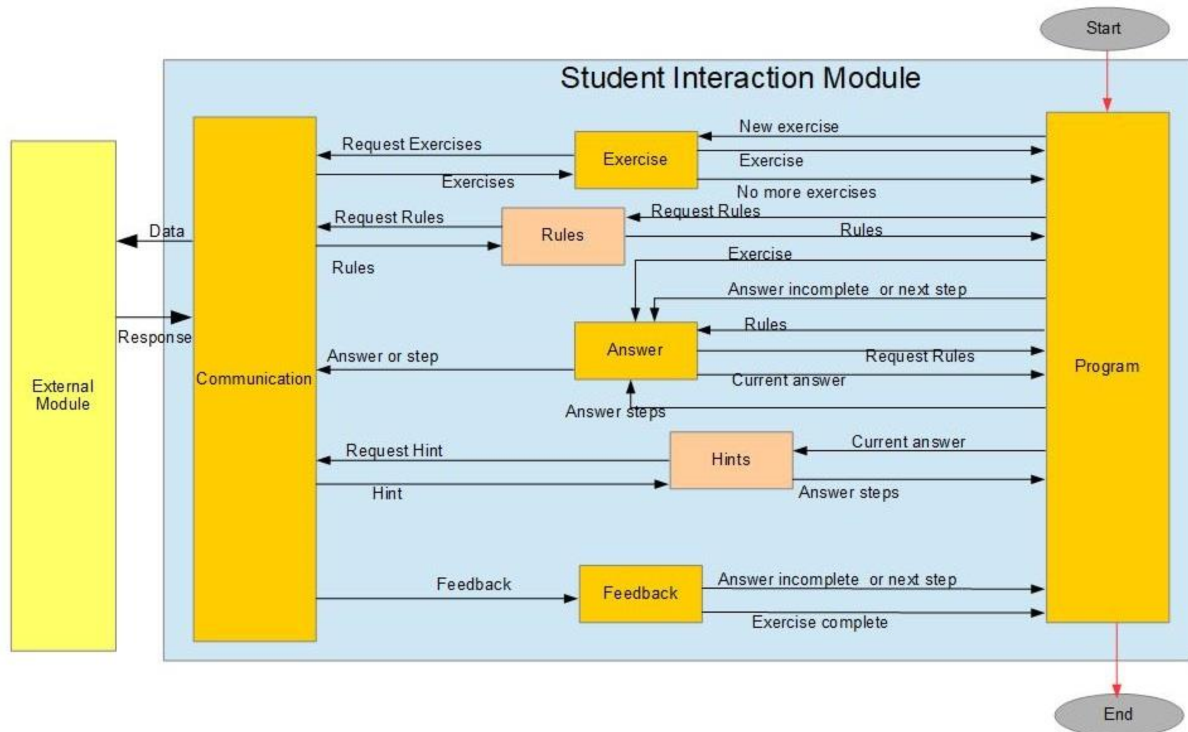


FIGURE 11: OVERALL VIEW (LEGEND SEE FIGURE 10)

5.3.4.3. Program

Program (see Figure 12) is responsible for the control and the general functions of the SIM and consists of ProgramControl, ProgramModel and ProgramView. The SIM starts with ProgramControl that controls the flow of the program. Except for CommunicationControl all control modules are connected with ProgramControl. The Forwarder of ProgramControl chooses the next step in the program that has to be done.

ProgramModel holds variables that are needed for the control of the program and can be queried by ProgramControl and ProgramView.

ProgramView is used for showing main page in which the information from the other views is shown and for general information that is not related to the other entities.

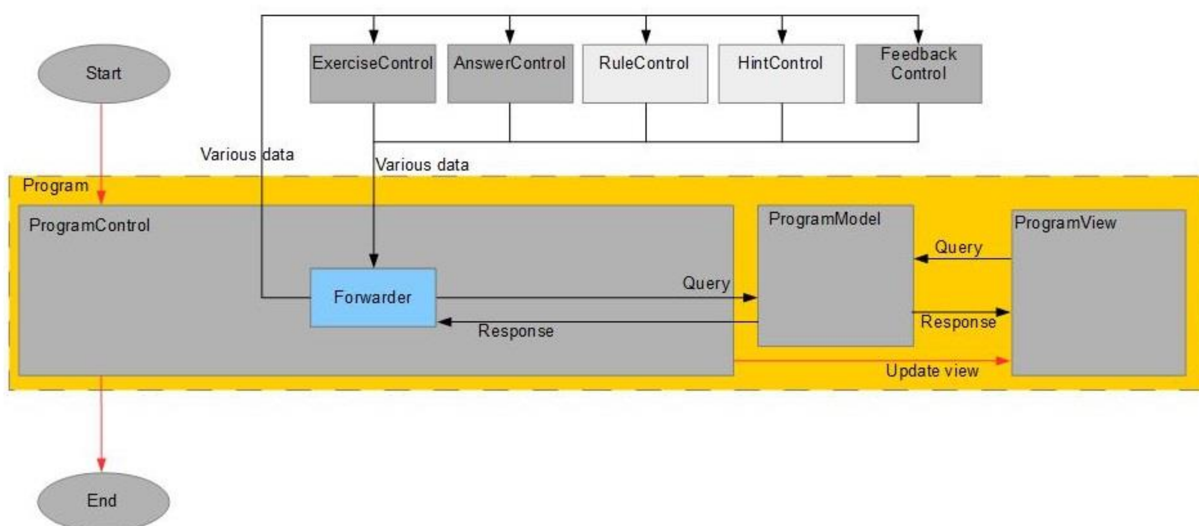


FIGURE 12: PROGRAM DIAGRAM

5.3.4.4. Exercise

Exercise (see Figure 13) consists of ExerciseControl, ExerciseModel and ExerciseView and is responsible for the selection of an exercise that has to be solved by the student.

In ExerciseControl the functions ReceiveExercise and ReceiveExerciseList are added to handle exercises and list of exercises retrieved by the communication modules from the external module.

In ExerciseView the function ExerciseView handles the presentation from the features ShowExercise, UserSelectExerciseControl and UserDefinedExerciseControl from the feature model and ExerciseControls supplies the controls (buttons, menus etc.) for those features.

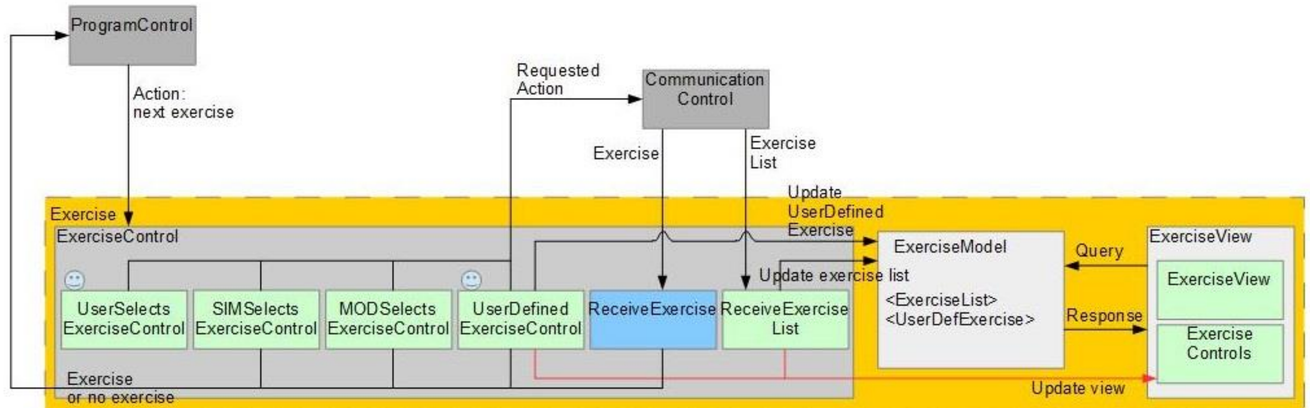


FIGURE 13 : EXERCISE DIAGRAM

5.3.4.5. Answer

Answer (see Figure 14) consists of AnswerControl, AnswerModel and AnswerView and is responsible for handling the answer of the student, storing it and presenting it.

In AnswerControl the functions RetrieveRules, HandleNextAction and HandleExercise are added. RetrieveRules is used to retrieve a set of rules that can be used when giving an answer. HandleNextAction is used to handle a next action from Hint or from Feedback. From Hint can this be a next step or the completion of the solution. From Feedback this can be storing the last step as a correct step or staying on the current answer because it was an incorrect step. HandleExercise is responsible for handling the current exercise which is presented within Answer.

In AnswerView the function AnswerView is responsible for the presentation of elements from the features InputAnswerFormat and AnswerTypeInput from the feature model. The function AnswerControls is responsible for the presentation of the controls from the feature AnswerControls from the feature model.

In AnswerModel the storage of the text of the exercise (ExerciseText), a list of rules that can be used when answering (CurrentRules), an input for the answer (AnswerInput) and the storage of the answer steps given by the student (CurrentAnswer), are added. AnswerFormat is also added to AnswerModel, this stores the definition how the answer can be given (input line or separate input box with parts of the answer, or case structure) this is missing in the feature model.

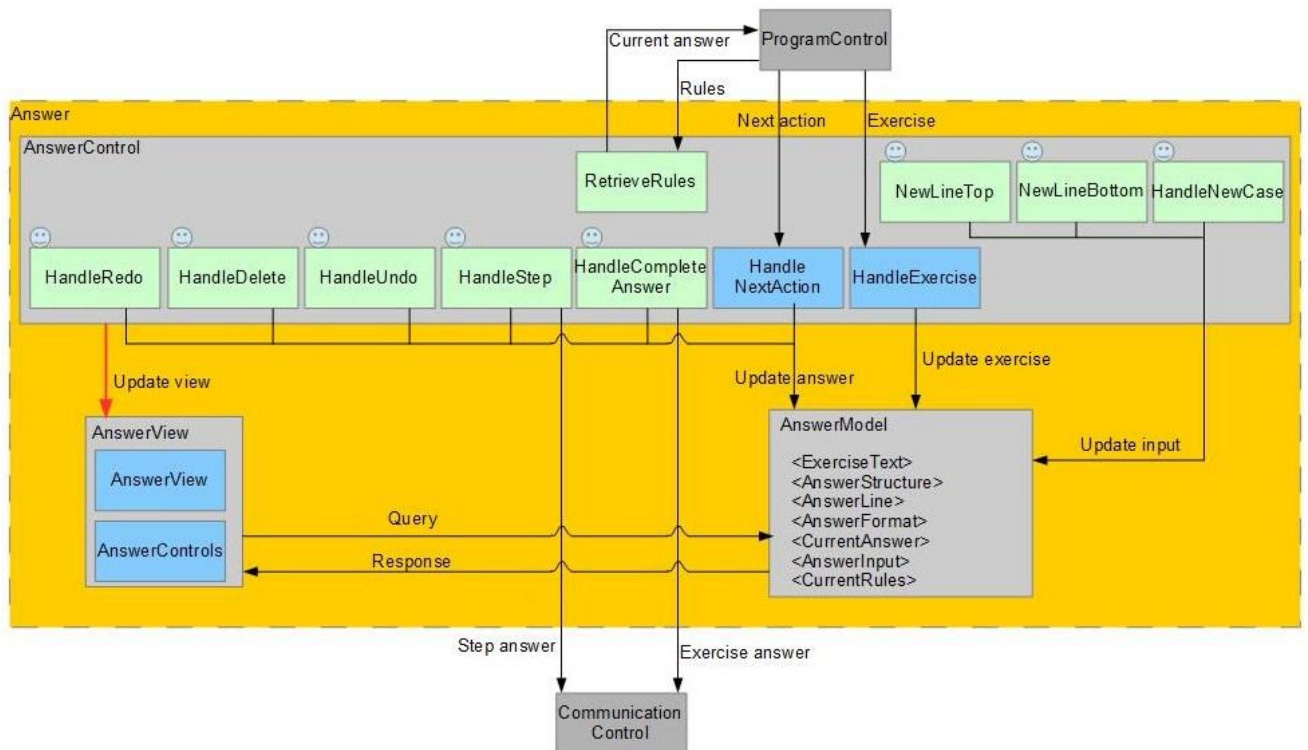


FIGURE 14: ANSWER DIAGRAM

5.3.4.6. Rule

Rule (see Figure 15) consists of RuleControl and RuleModel. Rule is responsible for providing the logic rules to answer, so that the logic rules can be used when answering exercises.

RuleControl in this model is the same as in the feature model.

In RuleModel RuleDefinition is added for the definition of one single rule and RuleList is added to store a set of rules using for every rule the definition of the rule.

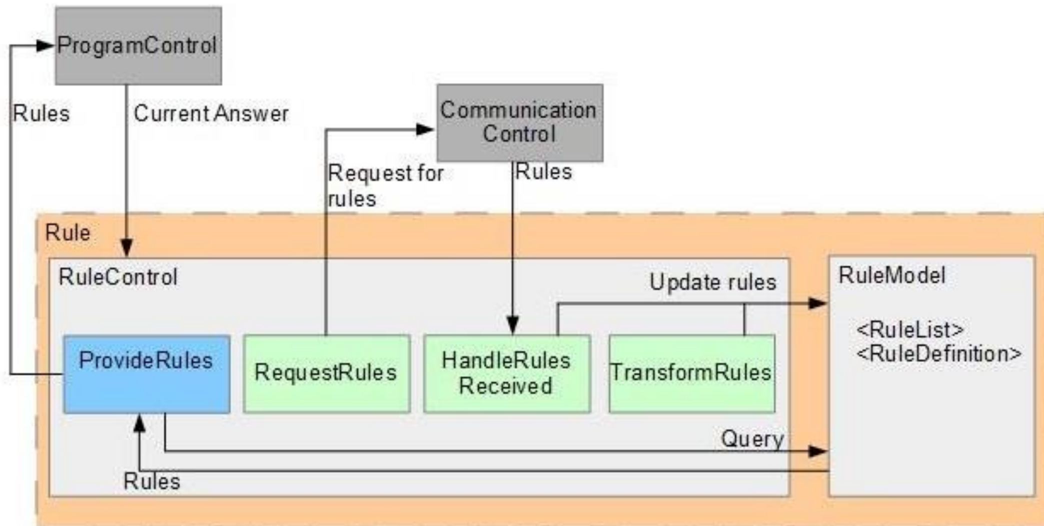


FIGURE 15: RULE DIAGRAM

5.3.4.7. Hint

Hint (see Figure 16) consists HintControl, HintModel and HintView and is responsible for providing hints on request of a student.

In HintControl functions are added to process the hints received from the external module; ReceiveNextStep for a next step as hint, ReceiveCompletion for the completion of the exercise as hint, ReceiveHint for a textual hint, ReceiveExample for an example as a hint and ReceiveSolution for the complete solution of the exercise as a hint.

In HintView both functions are the same as the features from HintView in the feature model, only the options mentioned in the feature model are here part of the functions.

In HintModel NextStep and Completion from the feature model are missing; these contain steps that have to be added to the answer and are forwarded by ReceiveNextStep and ReceiveCompletion from HintControl to AnswerControl.

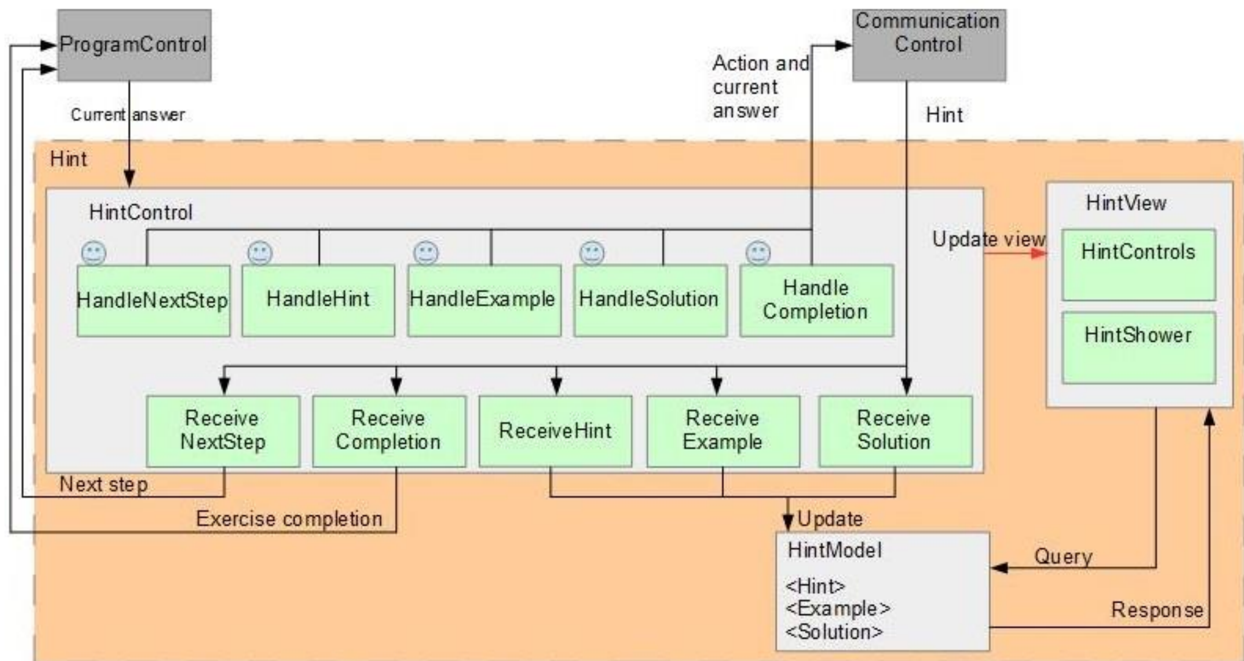


FIGURE 16: HINT DIAGRAM

5.3.4.8. Feedback

Feedback (see Figure 17) consists of FeedbackControl and optional FeedbackModel and FeedbackView and is responsible for handling received feedback from the communication modules.

FeedbackControl is the same as FeedbackControl from the feature model.

In FeedbackModel KR and FeedbackAction are missing; these two have affect the control of an answer and are forwarded by HandleFeedbackAction from FeedbackControl to AnswerControl.

In FeedbackView the options FeedbackMarker and FeedbackText from the feature model are part of the function FeedbackView.

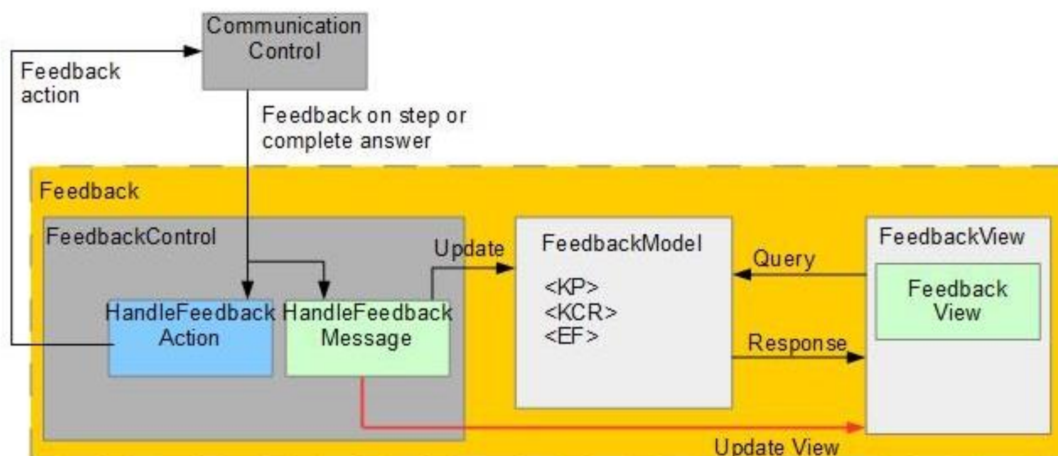


FIGURE 17: FEEDBACK DIAGRAM

5.3.4.9. Communication

Communication (see Figure 18) consists of CommunicationControl, CommunicationModel and CommunicationView and is responsible for handling all communication from SIM modules with an external connected module.

CommunicationControl and CommunicationView are the same as in the feature model.

In CommunicationModel ErrorMessages is added to store error messages during communication. The options of CommunicationFormat, CommunicationAction and CommunicationModule from the feature model are part of CommunicationFormat, CommunicationAction and CommunicationModule in this model.

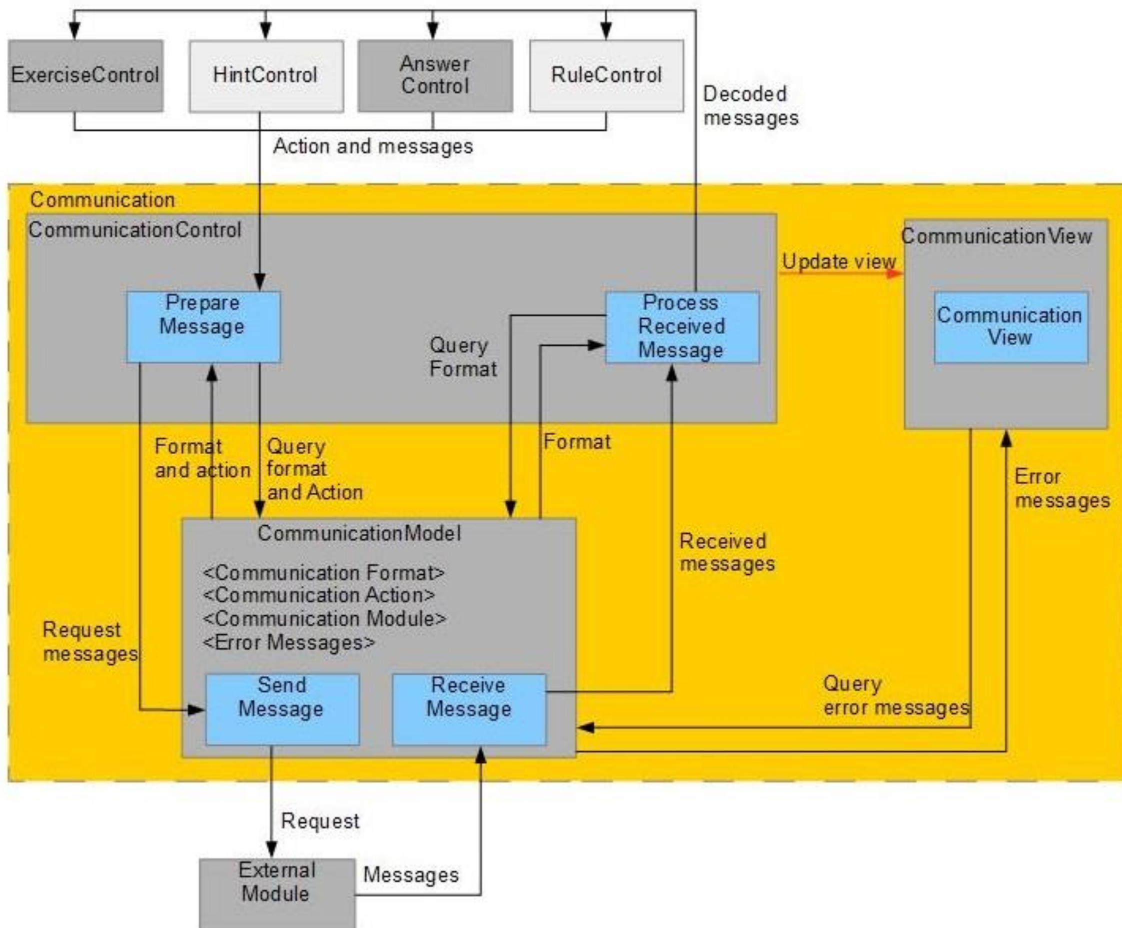


FIGURE 18: COMMUNICATION DIAGRAM

6. Proof of Concept

To prove that the designed architecture can be used to make the SIM a proof of concept of the SIM is built. ELMEX[10], an experimental frontend for IDEAS written in an older version of ELM version, is used as a base for this.

ELMEX is written to see and experiment how a front-end can be built with ELM. Is consists of five modules: Main for the start and initialisation of the program, Model to define the types that are used in the program, View for everything that is displayed to the user of the program, Controller for the overall control and the updates in the program and Ideas for everything that is related to the communication with the Ideas framework. The functionality of ELMEX is comparable to a part of the functionality of LogEx[11] and is limited to getting a random exercise from the IDEAS framework, which is presented to the user, the user can get hints and stepwise solve the derivation, whereby every step is checked by the IDEAS framework.

For our proof of concept we have updated this version to ELM version 0.19, done a redesign on the code to fit it in our architecture and added functionality to retrieve example exercises from the IDEAS framework and we have made it possible to offer more than one exercise to the student.

ELM is a functional programming language that compiles to JavaScript[12]. Because JavaScript can only be used in front-ends, ELM can only be used to program front-end application. For the proof of concept of SIM this is not a problem, because it is only a frond-end application.

Our proof of concept consists of ELM modules that are compiled to one JavaScript program. This JavaScript program is embedded in an index.html, to load it into the browser of a user. In the index.html bootstrap and font-awesome, both tools for web-layout, are made available for the JavaScript program.

In the technical reference we describe the program in detail. In the following subsections, we outline some aspects related to the general flow of the program and the modules of the program.

6.1. General flow of the program

This paragraph describes the general flow of our proof of concept. We describe the structure of the program, the usage of type definitions and variables and the way data from the external module is retrieved.

6.1.1. Structure of the program

An ELM-program has a specific structure. It starts with a main-function, this main-function holds the definition of the program, and it defines the initialisation function that should be executed once at start, the update function that should be executed after each update and a view function that should be executed after each update. The update and view function are executed automatically every time an update takes place. The updates are triggered by messages, and the loop of the execution of update and view is done until all messages are handled. The messages are generated by user actions and by functions in the program (see Figure 19). For example the student clicks on a button for the next exercise, this will result in a message, this message is handled by the update function and the function to retrieve the exercise from the external module is executed. The result of this call is send back with a message to the update function. Next a function that updates the variable that holds the exercise is executed and automatically the view function is executed, this view function

[10] https://ideastest.science.uu.nl/elmex/

[11] http://ideas.cs.uu.nl/logex/

[12] https://elm-lang.org/

has a part in which the exercise is shown. After this there are no more messages and the program will wait till the student gives input.

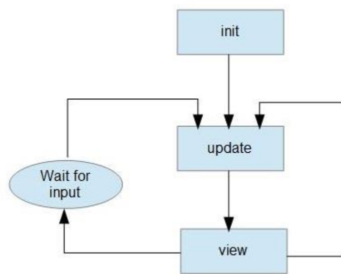


FIGURE 19 : FLOW DIAGRAM OF AN ELM PROGRAM

6.1.2. Variables and type definitions

Every variable in a ELM program is bound to a type definition, which defines the possible values of that type. A variable can hold a simple value or a list of values or a structure of different types of values or a list of such a structure. ELM is a functional language, so functions have no side effects. In relation to variables this means that a function cannot store the value of a variable, it can only calculate the return value of the function.

As far as we could see it is in an ELM program only possible to define two variables in the Main module: one to hold the values used in the program and one that is used for messages that are used to control the flow of the program. Other variables are only visible in the function in which those are defined.

The variable to hold the values is called model and is of the Model-type, which is a structure of variables. This model variable is initialized in init of the Main module and used in the update function of the MainControl module where the values are updated by the functions called by the update function.

The variable to hold the messages is called msg of the Msg-type. This variable is used in the update function of the MainControl module for the control of the flow of the program. If a function returns values that have to be handled by another function, the function returns a message with those values and the update function of MainControl calls the next function with the values as parameters.

6.1.3. Retrieving data

The IDEAS framework provides our proof of concept with exercises, hints and diagnoses. To retrieve data from the IDEAS framework our proof of concept every time uses the same construction (see Figure 20);

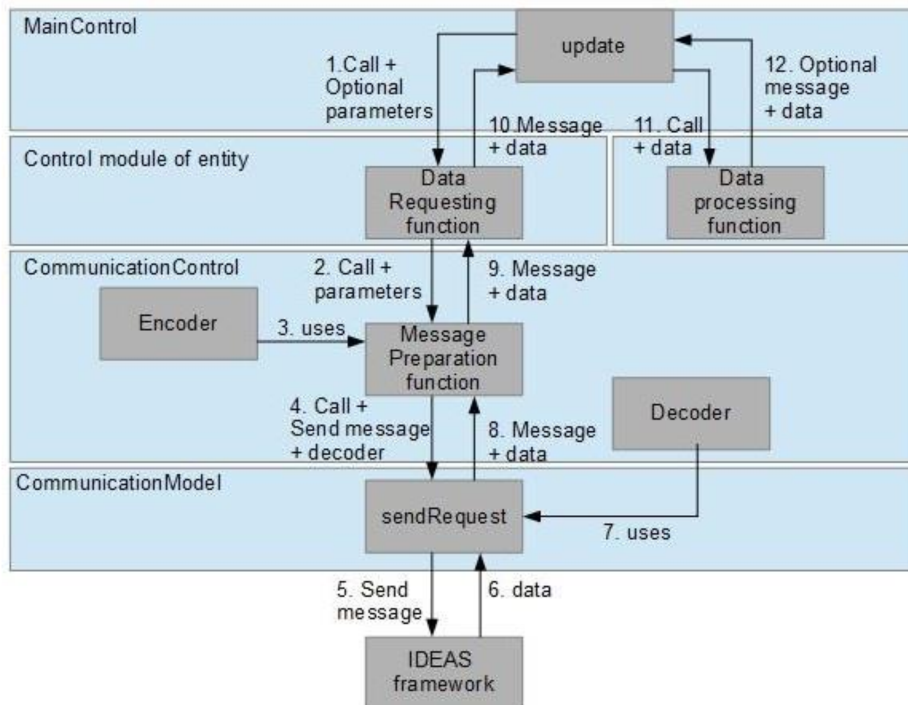


FIGURE 20: POC: DATA RETRIEVAL

1. A function belonging to an entity is called by the update function of MainControl.
2. The function of the entity calls a function in the CommunicationControl module.
3. The function in the CommunicationControl module encodes a message that has to be send to the IDEAS framework
4. The function in CommunicationControl calls the function sendRequest from the CommunicationModel module with as parameters the encoded message and a decoder .
5. The function sendRequest sends the request to the IDEAS framework
6. The function sendRequest receives data from it.
7. The data is decoded by the function sendRequest using the provided decoder,
8. The data is returned to the calling function in CommunicationControl.
9. The function in CommunicationControl returns the data with a message from the Msg-type to the calling function belonging to the entity.
10. The message and the data are returned to the update function of MainControl.
11. The update function uses the message to call the function that processes the data and uses the data as parameter. The function that processes the date can be of the same entity as the one that request the data from step 2 but can also be from a different entity.
12. After this the next step in the program is taken, which can be sending another request for data or waiting for input from the student (as shown in Figure 19).

6.1.4. Feedback and messages

In our proof of concept the modules related to Feedback; FeedbackModel and FeedbackControl are strongly related to the module for Messages; MessagesView because of a design error.

When programming the proof of concept in ELM we experienced that it was easier to combine messages related to hints, feedback and communication (communication error messages). In ELM everything related to the presentation is triggered by the defined view function in the Main module, which is executed automatically after the update function has been executed. ELM creates JavaScript that results in a HTML-page for the user, this HTML-

page is divided into divisions (<DIV> sections). We wanted to show the messages related to Hints, Feedback and Communication in the same division, because of this we defined one view module for the showing of messages and called it at first FeedbackView. Functions related to the control of this view we added to FeedbackControl and we defined a type Feedback in the module FeedbackModel for the messages. Afterwards we realized that MessagesView was a better name for FeedbackView, because it shows all kind of messages and not only feedback messages, but we did not extract the functions and the types related to messages from FeedbackControl and FeedbackModel.

6.2. Modules

In the next paragraphs we describe the modules of our proof of concept. We concentrate us on the most important modules and the most important functions of those; most of the internal functions and cleaning and initialisation functions are skipped.

We start with an overall view of the complete proof of concept (see 6.2.1), next we describe the structure of the type definitions (see 6.2.2). After that we describe the structure of the views (see 6.2.3). Finally, we describe the other modules (see 6.2.4 to 6.2.10).

6.2.1. Overall view of the proof of concept

Our proof of concept starts with calling the main module, a base module in which the structure of the program is defined and initialization of the variables msg and model is done. In this module the functions update from the MainControl module and view from the MainView module are defined. These two functions will be executed in a loop afterwards (see Figure 19). In the diagram of the overall view of the proof of concept (see Figure 21) the modules Main and MainControl are combined because we do not know exactly how ELM communicates between the Main module and the defined update and view function and it was not important to examine this more.

The function view of the MainView module uses the modules ExerciseSelectionView, AnswerView and MessagesView to present the view to the student (see the left-side of Figure 21).

The MainControl module calls functions in the modules ExerciseSelectionControl, RuleControl, AnswerControl, HintControl and FeedbackControl and functions in these modules except for AnswerControl can call functions in the module CommunicationControl (see Figure 21 centre and right-side). CommunicationControl calls functions in the module CommunicationModel to request all wanted data from the external module (see Figure 21 lower left corner).

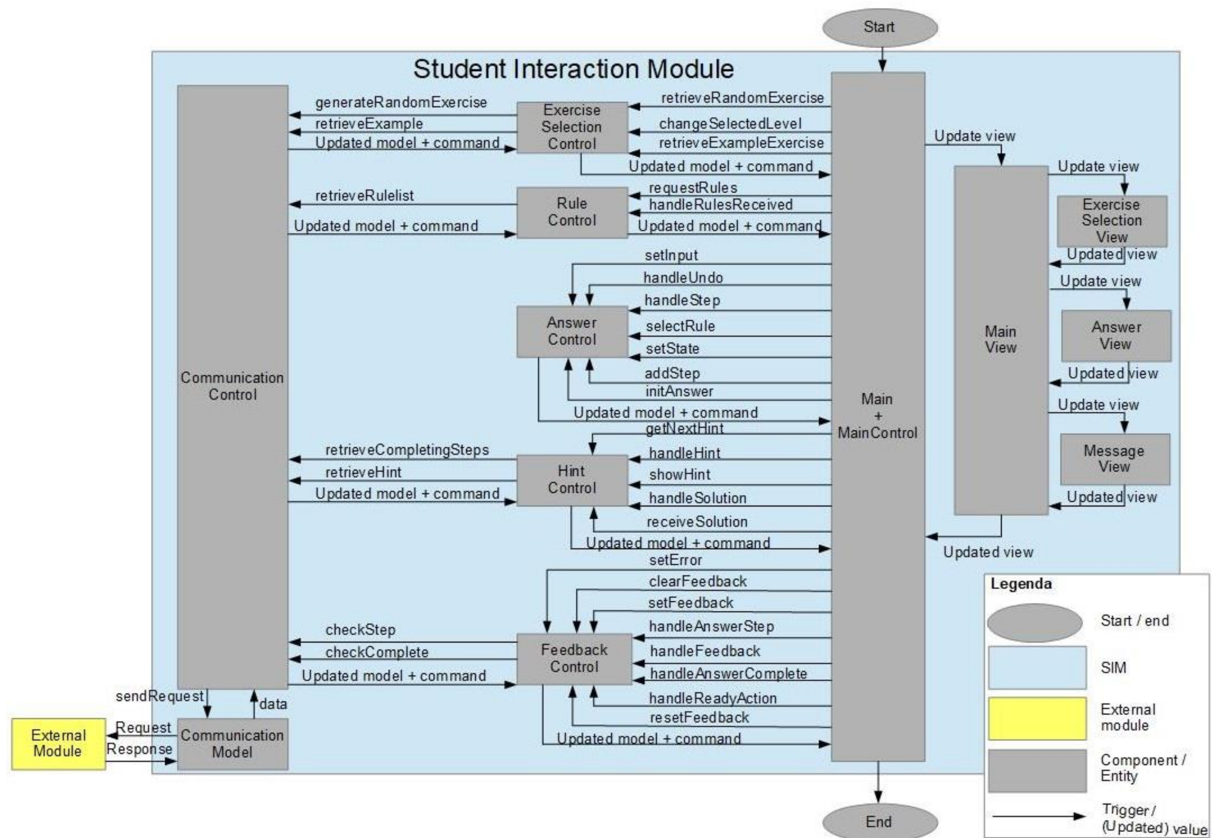


FIGURE 21: OVERALL DIAGRAM OF OUR PROOF OF CONCEPT

6.2.2. Type definitions in the Model modules

The Model modules in our proof of concept can have three functions:

- Defining types
- Defining constants and providing them to functions from other modules
- Communication with the external module

In this paragraph we look at first of these options; defining types. The other options are part of the description in paragraphs 6.2.4 to 6.2.10)

Our proof of concept is programmed in ELM, a functional language. Regarding types this means that all variables are from a pre-defined type or a type defined in the program. Correct usage of a type is checked at compile-time. Figure 22 presents the Model modules and the relations between them.

There is one issue with this model; for the type definitions we have looked at the types needed to communicate with the IDEAS Framework, we are not sure if the type definitions also can be used when another module is connected to our proof of concept. Maybe it was a better solution to make a special IDEAS-model module with the definition of all types and inherit those types in the other modules. Advantage of that solution would be that when another module is connected to our proof of concept there would be a higher chance that only the special IDEAS-model module has to be replaced by another module.

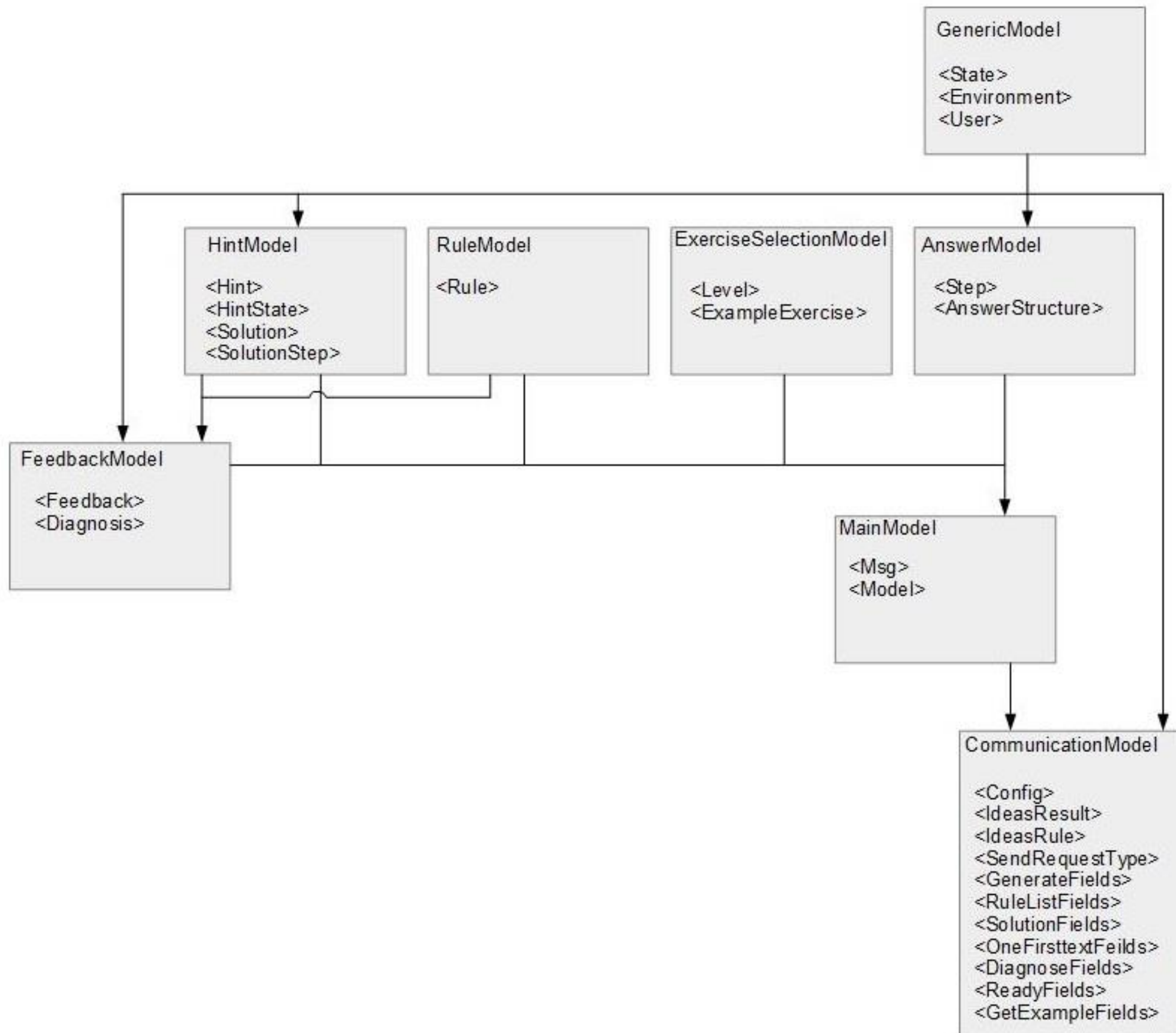


Figure 22: PoC: Type definitions in the model modules

MainModel contains the definitions of the types that are used in the entire program. Model is used to store values from and to supply values to all modules and Msg is used to control the flow of program. Both use definitions of types defined in the modules FeedbackModel, HintModel, RuleModel, ExerciseSelectionModel and AnswerModel. The real type definitions are encapsulated in those models and are hided for the MainModel.

FeedbackModel uses the type definitions of HintModel and RuleModel for the definition of the definition of the Diagnosis-type; the outcome of a diagnosis can contain hints and rules.

The genericModel contains types that are not related to one of the other entities, but are needed in definitions of types of other modules.

6.2.3. The view modules

The main view function, the function view from the MainView module is defined in the Main module. Figure 23 presents the relation between the view modules of our proof of concept.

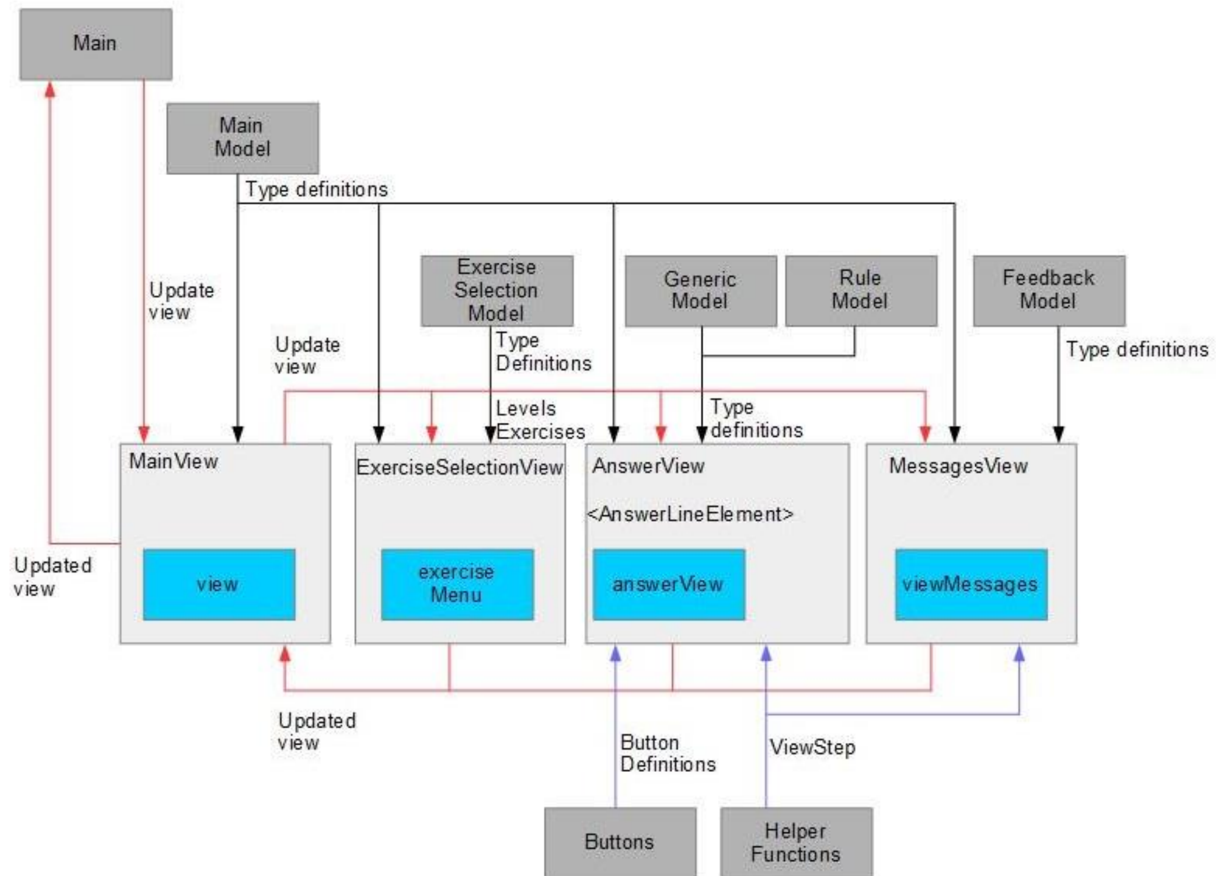


FIGURE 23: POC: RELATION BETWEEN THE VIEW MODULES

The function view of the MainView module calls every time it is executed the functions exerciseMenu from the ExerciseSelectionView module, answerView from the AnswerView module and viewMessages from the MessagesView module.

The function exerciseMenu presents the menu that is used by the student to select the exercise level for randomly generated exercises and to select the example exercise.

The function answerView presents the answering part of the page, which contains the inputs for the answer and all the control buttons provided by the Buttons module.

The function viewMessages presents all messages related to hints, feedback and communication to the student.

The HelperFunctions module is a supporting module and is used by AnswerView and MessagesView. It only has one function: viewStep that is used to present steps in the answer and to present steps in hints when asked for the completing steps.

The module AnswerView contains the datatype AnswerLineElement, this datatype is responsible for the definition of an answer line that is presented on the page and could not be added to the AnswerModel module because it would result in a Cycle Reference Error. Because this datatype is only used in AnswerView we have added this to the AnswerView module.

6.2.4. Main

Main (see Figure 24, use Figure 25 as legend) consists of the Main, the MainControl, the MainView and the MainModel module. The MainModel module is discussed in paragraph 6.2.2 and the MainView module is discussed in paragraph 6.2.3.

We are not sure whether Main or MainControl controls the loop between the update function of MainControl and the view function of MainView. In our diagram we have connected both to the Main module, but this can also be controlled by the MainControl module.

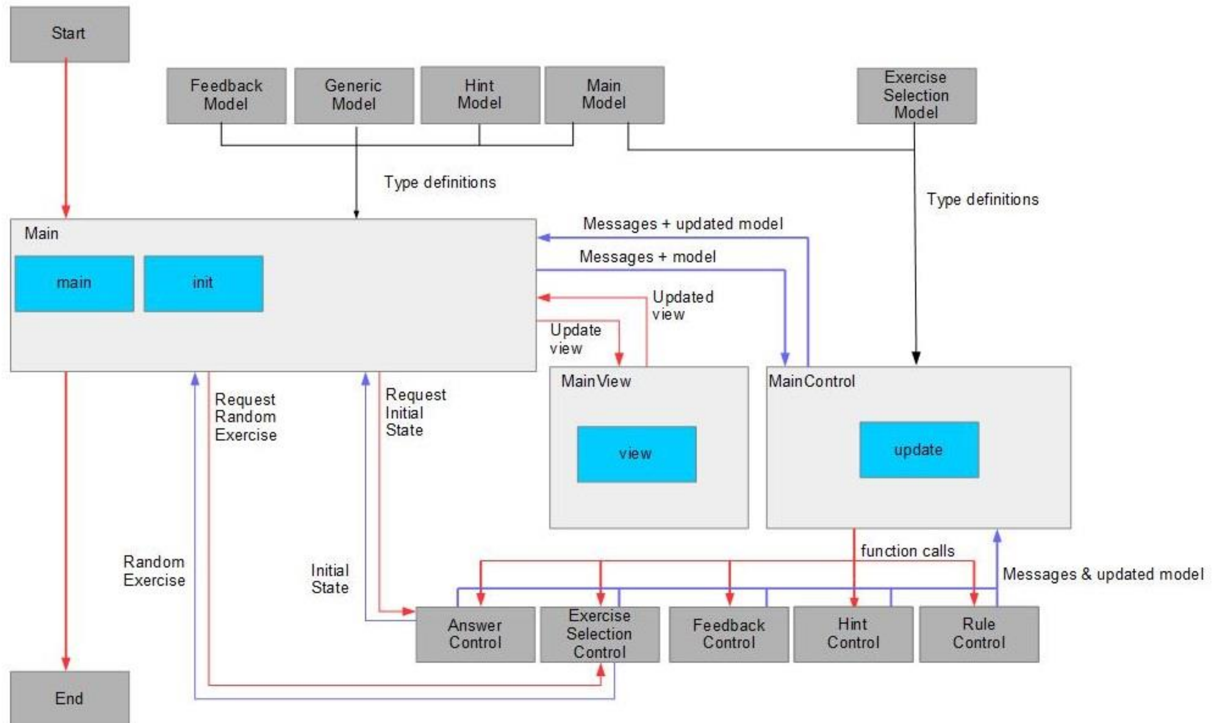
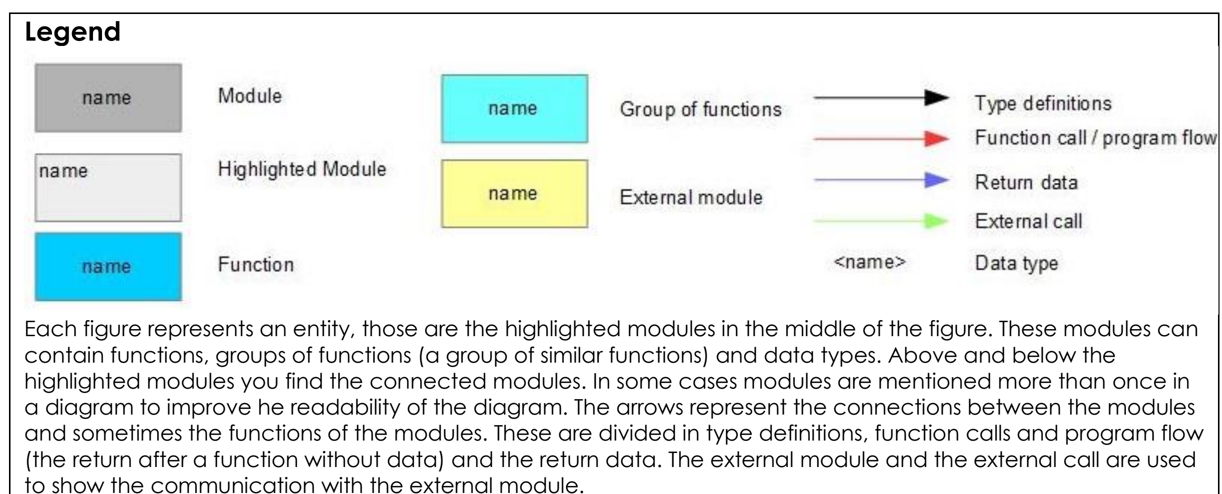


FIGURE 24: POC: MAIN DIAGRAM



Each figure represents an entity, those are the highlighted modules in the middle of the figure. These modules can contain functions, groups of functions (a group of similar functions) and data types. Above and below the highlighted modules you find the connected modules. In some cases modules are mentioned more than once in a diagram to improve he readability of the diagram. The arrows represent the connections between the modules and sometimes the functions of the modules. These are divided in type definitions, function calls and program flow (the return after a function without data) and the return data. The external module and the external call are used to show the communication with the external module.

FIGURE 25 : LEGEND PROGRAM DIAGRAMS

Main and MainControl

The main function of the Main module is responsible for the definition of the program. The init function of the Main module is responsible for the initialisation of the variables model and msg that are used in the entire program. The update function of the MainControl module controls the flow of the program.

6.2.5. Exercise selection

Exercise Selection (see Figure 26, use Figure 25 as legend) consists of the modules ExerciseSelectionControl, ExerciseSelectionView and ExerciseSelectionModel. ExerciseSelectionView has been discussed in 6.2.3 and is left out of the diagram.

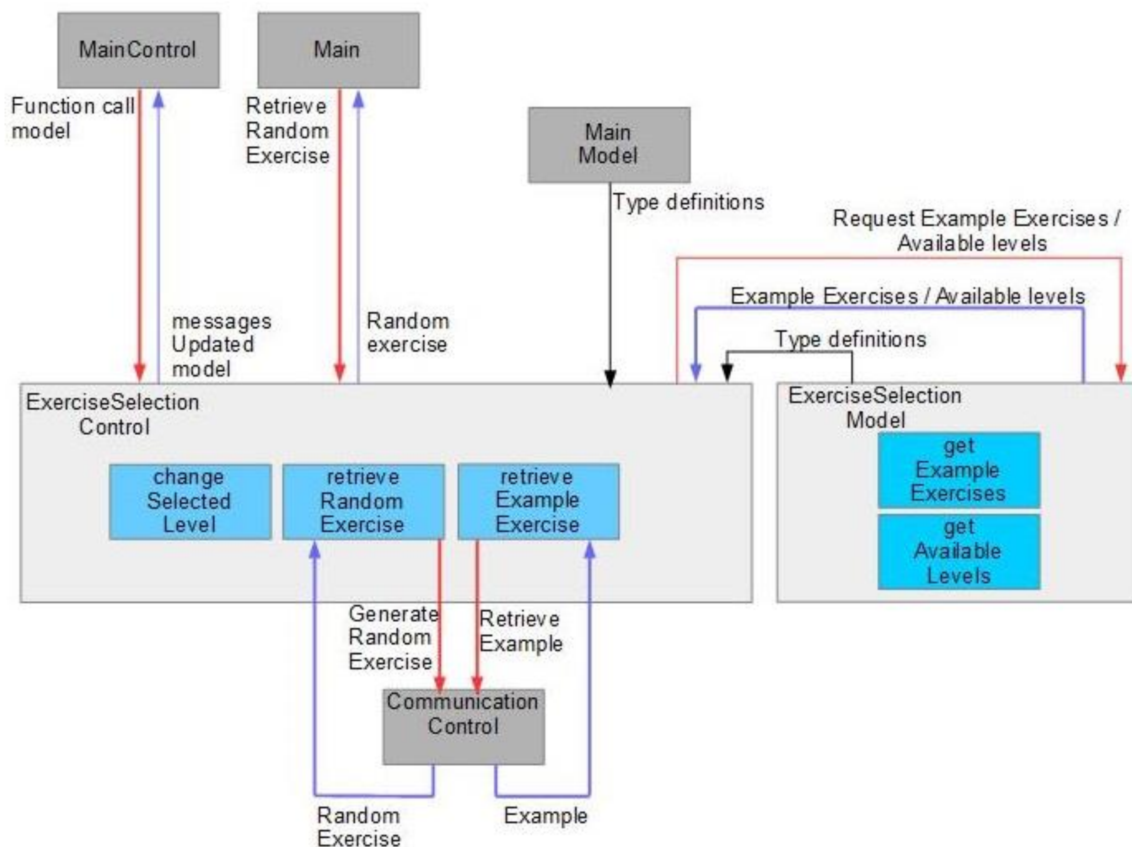


FIGURE 26: POC: EXERCISE SELECTION DIAGRAM

ExerciseSelectionControl

The retrieveRandomExercise function is responsible for the retrieval of random exercises, it uses functionality from the CommunicationControl module to actually retrieve the exercise (see 6.2.10). The retrieveExampleExercise function is responsible for the retrieval of a specific example exercise, it also uses functionality of the CommunicationControl module (see 6.2.10).

The changeSelectedLevel function is used to update the selected level for a next random exercise that is retrieved.

ExerciseSelectionModel

The functions getExampleExercises and getAvailableLevels are responsible for the retrieval of the possible example exercises and available levels of the proof of concept.

6.2.6. Answer

Answer (see Figure 27, use Figure 25 as legend) consists of the modules AnswerControl, AnswerModel and AnswerView. AnswerView is discussed in paragraph 6.2.3 and not shown in the diagram. AnswerModel is discussed in paragraph 6.2.2 and only shown a used module for used datatypes.

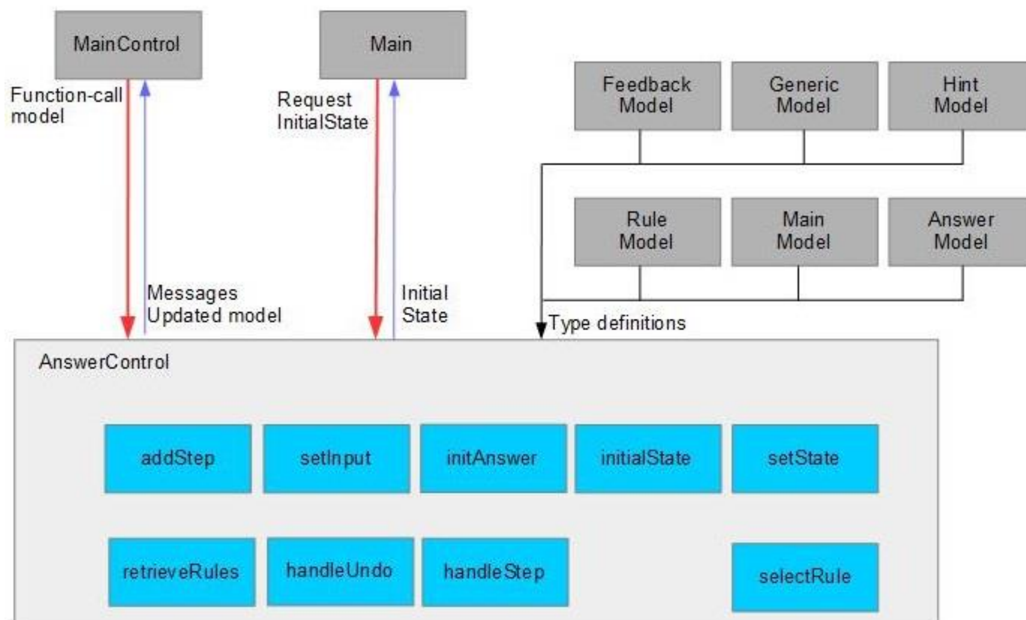


FIGURE 27: POC: ANSWER DIAGRAM

AnswerControl

The answer process starts with initialisation of some fields of the model variable and adding the exercise as first step of the answer.

Every step in an answer starts with retrieveRules, which triggers a process in Rules to make rules available for Answer. The student gives input which is handled by setInput or selects a rule which is handled by selectRule.

After a step the student can click a button to check a step of the answer, this is handled by the buttons module this triggers handleStep which checks whether the answer is complete and if it is complete forwards it to feedback for a diagnose.

The student can also click a button to have the complete answer checked, in which case feedback is triggered directly for a diagnose.

Depending on the result of the diagnose the process will return, if the step or the complete answer was incorrect, to the current answer line or, if the step was correct setState will add the values of the step to the list of given steps.

6.2.7. Hint

Hint (see Figure 28, use Figure 25 as legend) consists of the modules HintControl and HintModel. HintModel is discussed in paragraph 6.2.2 and is in this diagram only shown as source of input for datatype definitions. Everything related to the presentation of hints is part of the MessagesView module (see 6.2.3).

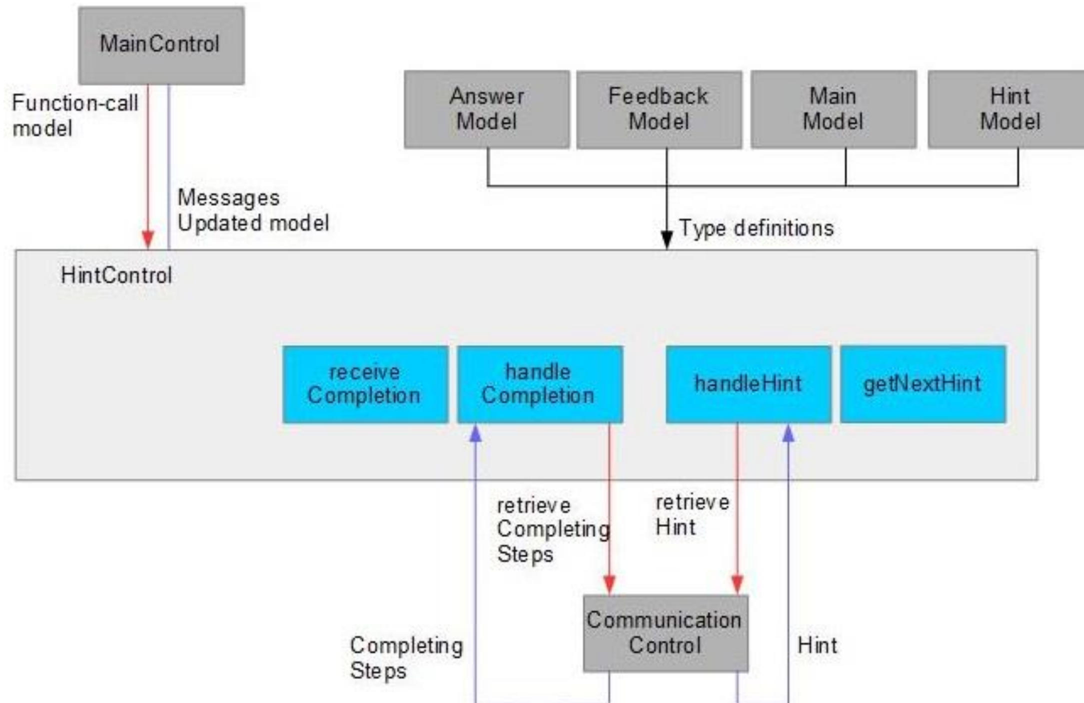


FIGURE 28: POC: HINT DIAGRAM

HintControl

After the hint button is pressed by the student the function handleHint is executed. This function uses functionality in the CommunicationControl module to retrieve a hint. The retrieved hint is send to getNextHint which determines the hint that has to be shown to the student.

After the completion button is pressed by the student the function handleCompletion is executed. This function uses functionality of the CommunicationControl module to retrieve the completion of an exercise as a hint. The retrieved hint is send to receiveCompletion to store the hint in the model variable.

6.2.8. Rule

Rule (see Figure 29, use Figure 25 as legend) consists of the modules RuleControl and RuleModel. RuleModel is already discussed in 6.2.2 and is in the diagram only shown as input for type definitions.

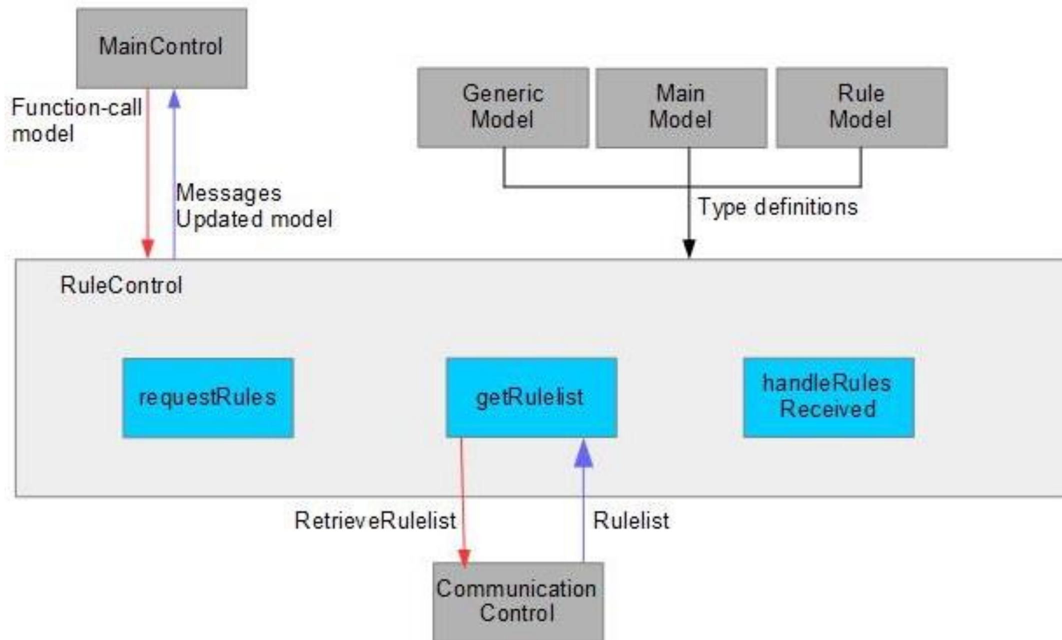


FIGURE 29: POC: RULE DIAGRAM

RuleControl

When rules are needed the function getRulelist is executed. This function retrieves a list of rules by calling a function in the CommunicationControl module. The requestRules function uses the getRulelist function but is needed because of a different return set of parameters. After both functions the handleRulesReceived function is used to store the list of rules in the model variable.

6.2.9. Feedback

Feedback (see Figure 30, use Figure 25 as legend) consists of the modules FeedbackControl and FeedbackModel. FeedbackModel is discussed in 6.2.2 and is in the diagram only shown as input for type definitions. There is not a view related to feedback; all the feedback messages are part of MessagesView (see 6.2.3).

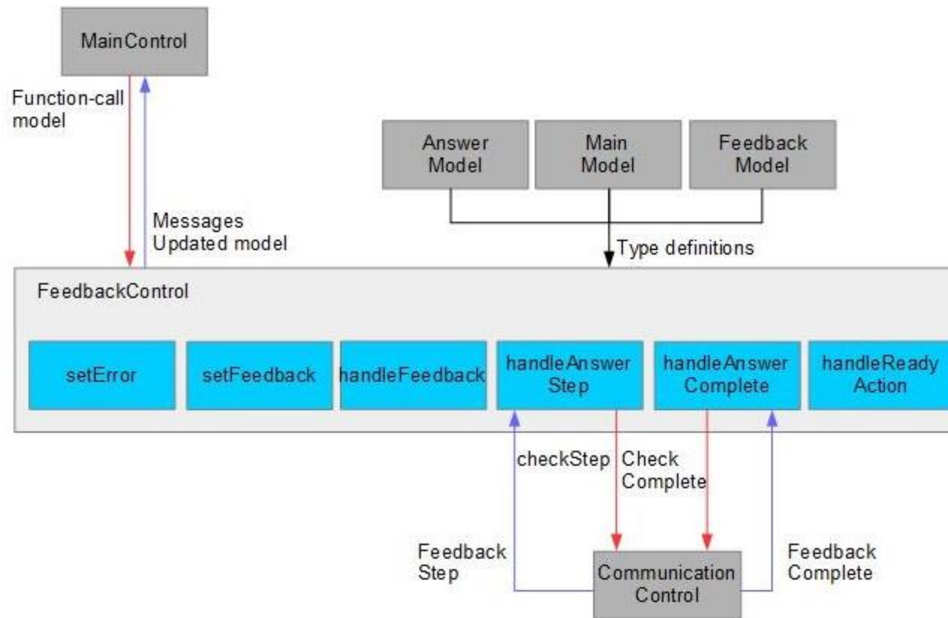


FIGURE 30: POC: FEEDBACK DIAGRAM

FeedbackControl

handleAnswerStep and handleAnswerComplete are both used to start the diagnosis by the connected module by calling CommunicationControl; handleAnswerStep is used for a step in the answer and handleAnswerComplete is used to check whether the derivation is complete. Both functions return a message with the result of the diagnose. The processing of the received data is done by the functions handleFeedback and handleReadyAction.

setFeedback and setError are used to put text messages in the feedback variable of model, setFeedback for a positive result and setError for a negative result.

6.2.10. Communication

Communication (see Figure 31, use Figure 25 as legend) consists of the modules CommunicationControl and CommunicationModel. The type definitions of the CommunicationModel are discussed in 6.2.2, in this paragraph we only discuss the functions of both modules.

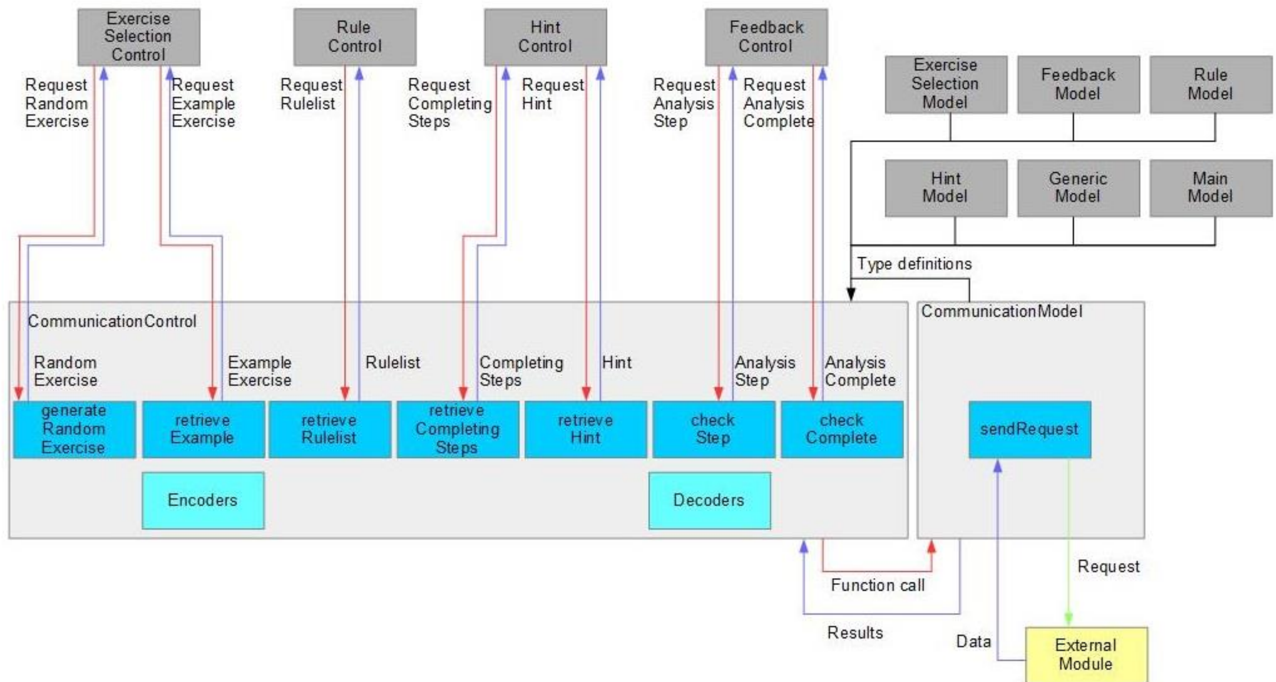


FIGURE 31: POC: COMMUNICATION DIAGRAM

CommunicationControl and CommunicationModel

The functions generateRandomExercise, retrieveExample, retrieveRulelist, retrieveCompletingSteps, retrieveHint, checkStep and checkComplete are called to retrieve data from and to have results checked by the connected module. These functions use the encoders to encode the messages and send those encoded messages to the sendRequest function of CommunicationModel that communicates with the connected module. The result of the communication with the external module is decoded with the decoders of CommunicationControl.

The module CommunicationModel has one issue that we could not solve; it exposes all the types and functions defined in it, although the function formatError is meant to be used only in CommunicationModel. For an unclear reason we could not do anything else than make everything visible.

6.3. Architecture vs proof of concept

In this paragraph the most important differences between the architecture of the SIM and the proof of concept is discussed. These differences are:

D.1. Features from the architecture missing in the proof of concept.
D.2. Functions of the proof of concept not mentioned in the architecture.
D.3. Features from the architecture split up in more than one function in the proof of concept.
D.4. Difference between the MVC-pattern in the architecture and the one in the proof of concept.
D.5. The function of the model modules in the architecture and in the proof of concept
D.6. The view modules related to messages to the student.
D.7. The presentation of buttons.
D.8. Differences in the communication modules.
D.9. The connection with the Communication modules.

These differences will be discussed in the sections below, in summary, they have the following causes and consequences:

CC.1. The architecture has more functionality than the proof of concept, because it must encompass multiple ITSs (D.1). In appendix A we describe in detail how the other features could be added to the proof of concept.
CC.2. The level of the architecture is too high to describe every possible functionality of an ITS (D.2). Since the differences between the proof of concept and the architecture are small, there is not a need for a change in the architecture.
CC.3. The connected module does not only affects the communication between the SIM and the connected module, but also the way how characteristics of entities are presented, asked for and checked by the SIM (D.6 and D.8). In our proof of concept this can be solved by adding extra modules with the elements related to the connected module (type definitions, format, checks on input etc.), that can be used by the other modules. In the architecture this kind of modules are part of Communication, this can be made part of an extra entity: ExternalModule that consists of modules with variables and functions that can be used by other modules.
CC.4. Some differences are related to implementation aspects (D.2, D.3, D.4, D.5 and D.7). Nothing has to be done with these differences; these are expected differences caused by programming in a specific programming language.
CC.5. Some differences are caused by errors made during programming (D.6 and D.9). These situations only affect the proof of concept and have no consequences for the architecture.

Only point CC.3 can have consequences for the architecture. This can improve encapsulation and data-hiding, which makes the entities Exercise, Answer, Hint, Feedback and Communication less related to the connected module. This improves the reusability of those modules.

Features from the architecture missing in the proof of concept.

Because the idea behind our architecture is that it should hold for more connected modules and our proof of concept is only build for one specific situation, not all the features of the architecture were needed in the proof of concept. In appendix A we describe in detail the missing features and how these can be added to our proof of concept if needed. Adding the features as functions to our proof of concept is not a problem.

Functions in the proof of concept not mentioned in the architecture

Our proof of concept has functions that are not mentioned in the architecture. These are functions related to implementation aspects such as getters, initialisation and clear functions. This was expected because the architecture is of a higher level than the proof of concept. It was expected that this kind of functions had to be added to our proof of concept.

There are also functions related to functionalities from the connected module that were not taken into consideration when making the architecture. In our proof of concept this applies for exercise level. Exercise level is in our architecture a characteristic of an exercise.

Features from the architecture split up in more than one function in the proof of concept.

In some cases it was needed to split up features from the architecture into more than one function in the proof of concept. An example of this are the view functions; in the architecture only one view function is defined, but in the proof of concept this has been split up in several functions. This is an implementation aspect caused by the higher level architecture as proof of concept. It was expected that this would happen.

Difference between the MVC-pattern in the architecture and the one in the proof of concept.

The architecture is based on an MVC-pattern in which the views are triggered by functions from the corresponding control modules. Our proof of concept is based on ELM where the main view function is automatically updated after the main update function has been executed. To update the separate views they have to be called from the main view function. Compared to programming in a traditional language in which every action has to be programmed separately, the ELM code is easier to program because there are no commands needed to update the views.

The function of the model modules in the architecture and in the proof of concept

The functions of the model modules in the architecture are: to store data and provide that data to other functions within the same entity and to communicate with connected modules outside the SIM. The functions of the model modules in the proof of concept are: to define the types, to store constants and provide the values to other functions within the entity and to communicate with connected modules outside the SIM. The model modules in the proof of concept do not store data from variables other than constants, which is part of the functionality in the architecture.

In our proof of concept we use the variables model of the Model-type and msg of the Msg-type to store the data; model is used to store values and msg is used to store messages with optional data belonging to those messages. These are variables of which the types are build up from types defined with the entities and can be used by the entire program (see 6.1.2). As far as we could see we could not use more variables in ELM, except for variables that are only available within functions. The usage of types of the entities in the Model-type can be improved by defining interfaces in the entity models; this way there will be less coupling between the MainModel module and the other model modules. The messages in the Msg-type can maybe be grouped in messages groups per entity, which would make them interfaces as well. This would also lead to less coupling between the model modules and it

would lead to less coupling between the MainControl module and the other control modules, but we have not done enough research to know if this will work.

Many definitions of the used types in our proof of concept are related to the IDEAS Framework. In paragraph 6.2.2 we suggest to add an extra model module with all the definitions that are related to the IDEAS-framework. This extra model module can be used as a façade for incapsulated definitions of the IDEAS types by the other model modules. This would make the coupling between the modules higher, but it would hide the details of how everything is stored in IDEAS for the other modules. IDEAS is one of the possible connected modules a better name for this extra module is: ExtenalModule.

The view modules related to messages to the student

The HintView, FeedbackView and CommunicationView modules from the architecture are in the proof of concept combined in the MessagesView module (see 6.1.4). The problem here is that the functions and the types related to the messages are combined in FeedbackControl and FeedbackModel of the proof of concept. This error causes a lower cohesion within the two modules. Below we describe how the presentation of the messages in our proof of concept works and which changes need to be made to correct our error.

> Feedback type; type used to store values of the messages. NoFeedback; show no message, ErrorFeedback; show an error message from Communication or Feedback, ReadyFeedback; show a message that a derivation is complete, HintFeedback; show a hint message of one line, SolutionFeedback; show a hint that consists of answer steps. This type definition should be part of MessageModel and called Message.
>
> The functions clearFeedback and resetFeedback of the FeedbackControl module; are both functions to clean the messages, resetFeedback Is the function that cleans the feedback, in our proof of concept this function is used when a new exercise is loaded or by the clearFeedback function. The clearFeedback function is used when the student wants to remove the message. The difference between the two functions are the return parameters, resetFeedback only returns a model, whereas clearFeedback returns a model and a message. resetFeedback turns the value of the Feedback to NoFeedback. These two functions should be moved to MessageControl.
>
> Messages are set to be shown by functions of HintControl, FeedbackControl and Communication control by a command *feedback* = flowed by a type of the Feedback type and variable that holds the message that has to be shown (this could also be a value), for example: *feedback = FM.ReadyFeedback feedbacktext*.
>
> The function viewMessages of the MessagesModule is used to show the messages. For every type of the Feedback type it calls a function that presents the feedback.

The functions called by viewMessage of the MessagesModule have duplicate code. This can be optimized by defining two functions: one for presenting single-line messages and one for presenting formulas and rules, that both have the class as a parameter (the class is used to determine the colour used to present the message).

Another improvement is to change the types of the Feedback type into types of messages that have to be presented (NoMessage, SingleLineMessage and StepMessage); this would make the types generic.

Making a separate HintView, FeedbackView and CommunicationView is a small step after the above improvements are made; all three would use the generic types and functions to present messages and the presentation could be done in separate <DIV> tags within the current <DIV> tag.

The presentation of buttons

In our proof of concept the buttons are presented as one line of buttons within AnswerView. In the architecture the buttons are part of the ExerciseSelectionView, AnswerView and HintView. The operation of the buttons is described below.

> All functions to present the buttons use the function actionButton from the module buttons. actionButton is a factory to make buttons.
>
> The buttons are first grouped per entity in the functions getAnswerButtons, getHintButtons and getExerciseSelectionButtons. These functions are combined in the getAllButtons function that is called from AnswerView to show the buttons.

Advantage of this solution is that all functions related to buttons are concentrated in one module with only a relation to MainModel for the definition of the types, but this is also a disadvantage; it makes it harder to delete optional functionality and the buttons module is more related to a specific implementation of the SIM.

We have chosen for this solution because adding the buttons to the model modules of the entities gave Cycle Reference Errors (the MainModel module uses the model modules, but the button functions need the types form the MainModel module).

Another option is to make entity-related modules for the buttons with an extra module for the generic actionButton function (it is not possible to combine this function with the getAllButtons function, because that will result in a Cycle Reference Error). In our proof of concept this option would result in 5 small modules.

In our proof of concept with all buttons in one line our solution is good to handle, but if the presentation would be divided into sections per entity with the related buttons the other option can be easier.

Differences in the communication modules

The features SendMessage and ReceiveMessage from the architecture are in our proof of concept combined in the function sendRequest. The functions generateRandomExercise, retrieveExample, retrieveRulelist, retrieveHint, checkStep and checkCompletion of our proof of concept are each a combination of the features PrepareMessage and ProcessReceiveMessage from the architecture and they use the Encoders and Decoders.

Because the functions generateRandomExercise, retrieveExample, retrieveRulelist, retrieveHint, checkStep and checkCompletion have the same construction; they all use a encoder to make a message that is send to sendRequest and they all use a decoder to transform the received data to usable data for the proof of concept. The only differences are the encoders, the decoders and the return messages. We expect for these functions a factory pattern can be used. This would result in one function that is responsible for preparing the message that is send by sendRequest to the external module and transforming the data that is received to data that can be used by the SIM.

The encoders and decoders are in our proof of concept part of CommunicationControl. Our encoders and decoders are functions that transform messages to and from the IDEAS framework format. In the architecture the encode and decode functions are part of PrepareMessage and ProcessReceivedMessage and they use a communication format that is stored in the CommunicationModel Module. We did not have enough knowledge of ELM to program it this way. Our proof of concept communicates with the IDEAS Framework, which does not use a standard communication protocol. We think that the encoders and decoders should be moved to a IDEAS related module, that is used by the CommunicationModel module. Doing this gives the possibility to use other similar external Modules without changing

the CommunicationModel and CommunicationControl modules. For a good implementation of the communication using standard protocols (see 5.3.3.3.1) more study has to be done on this subject.

The connection with the Communication modules

Our proof of concept has two differences with the architecture related to the connection with the Communication modules; the ExerciseSelection modules do not handle the result that is returned from the Communication modules, this is done by the setState function of AnswerControl and the Answer modules do not send the answer or a step of the answer for diagnosis to CommunicationControl, this is done by the functions handleAnswerStep and handleAnswerComplete.

The difference with ExerciseSelection is made because the exercise is used as the first step of the answer. We could also receive the exercise and first store it into a field of model in an extra function of ExerciseSelectionControl and retrieve the exercise from that field in AnswerControl, this would have made the ExerciseSelection part more independent from Answer.

The difference with the diagnose of an Answer is made to concentrate the send and receive functions into one control module, with the intention to combine these two functions into one function, to reduce the number of messages handled by the update function in the MainControl module. The problem with this solution is that FeedbackControl has knowledge about an answer in this construction, which was not the case in the architecture.

7. Flexibility of the architecture

In this paragraph the flexibility of the SIM architecture is examined, which is RQ. 3 of this study. The SIM architecture should be able to support all kinds of Logic ITSs based on a four component architecture. It is impossible to check if every existing Logic ITS can use the SIM architecture, and even if this was possible there will be new ITSs developed. That is why we look at the flexibility of the SIM architecture; if it is flexible, it is more likely that the SIM architecture can be used for other Logic ITSs as well. This is done by looking at other kind of ITSs and discussing how the SIM architecture can be used to support those ITSs. The SIM architecture is developed for supporting Logic ITSs, but if it is flexible enough to support another kind of ITS, it probably can also support other Logic ITSs. Two kind of other ITSs are discussed: ITSs for mathematic exercises and ITSs for programming languages.

ITSs for mathematic exercises

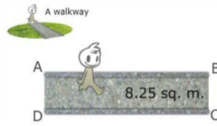
Mathtutor ("Mathtutor," 2020) is an ITS for mathematics. The SIM concentrates on logic, by looking at a mathematic ITS we think that if an ITS can support most of the mathematic exercises it also can support most of the logic exercises.

Mathtutor contains multiple tutors in various categories. Because of the many options we only describe three of those, as far as we could see, all tutors have a similar approach. We use 7.52 Solving Linear Equations with Parentheses of the category solving equations (see Figure 32), 6.16 Area of Polygons of the category Area, Perimeter, Circumference (see Figure 33) and 8.26 Interpreting Box-and-Whisker Graphs of the category Box and Whisker Plots (see Figure 34).

FIGURE 32: 7.52 SOLVING LINEAR EQUATIONS WITH PARENTHESES

6.16 Area of Polygons: Activity 3 of 28

You are designing a walkway from the garage to your back porch. You have enough cement to cover 8.25 square meters. To the right are some possible widths for the walkway. What width would give you the longest walkway?



(1) The width of the walkway 0.75 meters, how long will the walkway be?

(2) The width is 0.6 m, what will the length be?

(3) You want a skinny walkway at only 0.3 m wide, how long could the walkway be?

	Area of the walkway	Width of the walkway	Length of the walkway	Longest walkway?
Unit	sq. meters (sq m)	meters (m)	meters (m)	
Diagram Label	---	BC	AB	
1				○ 1
2				○ 2
3				○ 3

? Hint

Done

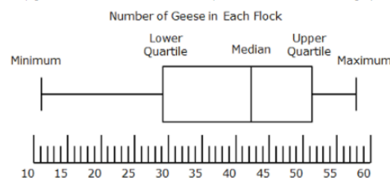
Previous Next

Decimal Division
Decimal Multiplication
Identify expression for the perimeter
Identify expression for a segment
Find perimeter of a parallelogram
Identify base segment
Identify segment adjacent to the base

FIGURE 33: 6.16 AREA OF POLYGONS

8.26 Interpreting Box-and-Whisker Graphs: Activity 1 of 8

An environmentalist was studying how geese gather together in flocks to migrate south for the winter. Over the period of a week, she counted how many geese were in each flock that flew by overhead and created this graph:



Please use the box-and-whisker plot to answer the questions below.

The largest flock had 58 geese.

25% of the flocks had at least 51 geese.

50% of the flocks had at least 42 geese.

75% of the flocks had at least 29 geese.

The smallest flock had ____ geese.

? Hint

Done

Previous Next

Identify value given a quartile
Identify value for maximum or minimum
Identify value given an end of a range

FIGURE 34: 8.26 INTERPRETING BOX-AND-WHISKER GRAPHS

The exercises in Solving linear equations with parentheses are divided in two parts. The first part, in which the formulas and the rules have to be given, looks like the exercises in our proof of concept; the input is a formula that has to be entered until the answer is complete. the second part, in which only the final answer must be given, is different from what we had in the proof of concept, but could be solved as a formula without a rule. The exercises in Area of Polygons use a table in which the answers have to be entered, this is different to our proof of concept because the proof of concept does not have any support of tables at this moment. The exercises in Interpreting Box-and-Whisker Graphs use graphics in the text of the exercise, this is not supported in our proof of concept at this moment.

All three have a part in which the exercises and the answers are given, a part in which help and text feedback is given and they all three have a part in which the KP feedback is given. The KP feedback is not part of our proof of concept.

The part in which the exercises and the answers are given fits within our architecture. The showing of the exercise and input of answers is part of the answer modules in our architecture. The showing of images can also be done in AnswerView, but probably a supporting module for showing the images that is used by AnswerView is wanted.

Mathtutor has two separate areas in which hints and feedback are shown. In the SIM architecture there are two separate view modules for these messages, but because the two

areas are presented next to each other, they can also be combined in one module like we did in our proof of concept.

Mathtutor uses the jQuery ajax method. This method send data to a specified URL and decodes the return data with a specified decoder. This is basically the same as the construction we use in our proof of concept in the Communication modules.

Our conclusion is that the parts of Mathtutor we have seen fit in our architecture, but can lead to some specific extra supporting modules.

ITSs for Learning a programming language

Ask-Elle (Gerdes et al., 2017, p.) is an ITS for learning Haskell that uses the same IDEAS framework as we did for our proof of concept. We look at Ask-Elle because the learning material is not related to mathematics, if Ask-Elle can be supported by our architecture as well, there is a bigger chance that our architecture can support all kind of logic exercises.

Ask-Elle (see Figure 35) has an exercise selection, a part in which the current exercise is shown, an editor part for entering the answer and a Help part to ask for help and to give feedback (see Figure 35). The exercise selection fits in the ExerciseSelection modules we have in our architecture. The current exercise is in our architecture part of the Answer module; but in our proof of concept this line is shown as part of the answers given by the student.

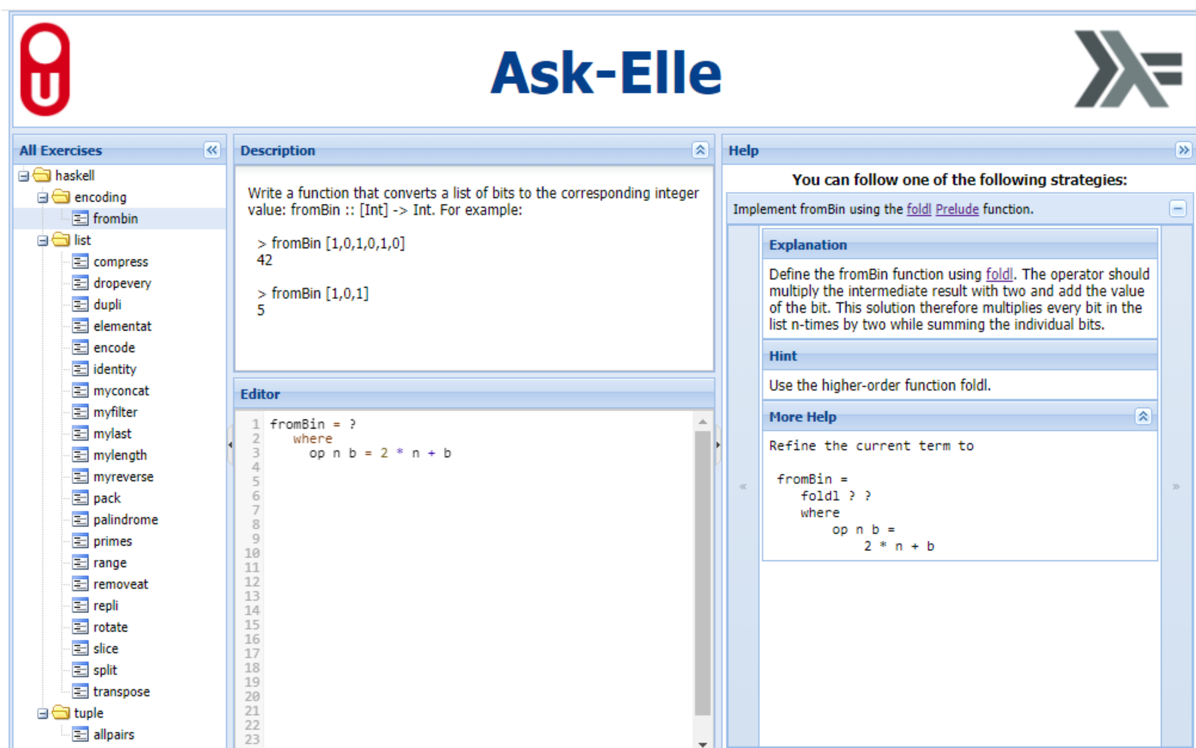


FIGURE 35 : ASK-ELLE

The editor part is in the architecture part of the Answer modules, but in this situation the answers that are already given by the student and the import lines are the same. This fits within the architecture, but the implementation is different from the one we have chosen for our proof of concept.

The answers in Ask-Elle are the program that is developed. Ask-Elle uses a special editor for this and the complete program is send to the connected module for diagnose and hints. In our proof of concept we used a step-wise answer, where every step consists of one entry line.

Our architecture also supports constructions in which more than one entry line is part of an answer. Therefore it has the possibility to use an answer-structure in which entry lines can be defined. This is meant to define case-constructions that are used with inductive logic, but this can also be used for the definition of a program with lines of code. The special code editor can be a separate module that is used within the Answer modules.

Ask-Elle has a part that presents hint and feedback messages, both are supported by the SIM architecture. The hint messages can have several layers, the SIM architecture does not show such a detail about hints; it only shows that hints can be used and it offers the possibility to retrieve more hints, how the hints are shown and if a hint consists of more than one level is seen as an implementation aspect.

Our conclusion is that Ask-Elle can probably be built based on our architecture, but the response part uses the answer structure in a way that is not intended, this is not a real problem.

8. Discussion

In this chapter the proof of concept of the SIM is compared with the SIM architecture and the differences between both is discussed. Also the flexibility of the SIM architecture is discussed by looking at two non-logic ITSs and describing to which extent the interaction part of those ITSs can be designed with the SIM architecture and in the last paragraph this study is discussed.

8.1. Limitations of the study

A narrow basis for this study

This study is based on just a few studies in which requirements and quality guidelines for the front-end of an ITS were mentioned and only one interview with a stakeholder. This is a narrow basis for the study on the architecture.

It is possible that this basis was too narrow and there are more stakeholders for some of the requirements and quality guidelines. It is also possible that requirements and quality guidelines have more attention points then mentioned in this study. Nevertheless. it is expected that the architectural drivers on which the architecture are examined remain the same, because these are valid for reusable systems in general.

The depth of the study

The depth of the study is limited. A deeper study of the relation between Hints and Feedback can lead to an outcome in which both are combined into one entity or an extra messages entity that is related to both.

In the study the communication between the SIM and the connected modules and the effect of the connected modules for the SIM, which can lead to the need of an extra entity related to the connected module, is already mentioned. A deeper study on this can also lead to the need for more than one extra module; one for the connected module and one or more related to the communication of the connected module with the SIM.

Proof of concept is very limited

The proof of concept is very limited; the developed SIM only contains a small part of the possible functionality, it does not follow the architecture completely, the architectural drivers maintainability and compatibility are not proven by the proof of concept and it connects only to a knowledge module and is therefore only a part of an LE and not on an ITS.

For all these arguments a written explanation has been provided with possible solutions. For the missing functionality, it is described how this can be added to the proof of concept. For the deviations from the architecture, it is described how this could have been built differently. Maintainability is covered by using proven architectural concepts that improve the maintainability. The difference between an LE and an ITS is seen from the SIM only another module that is used for communication. Only the evidence for compatibility is poor; during the construction of the proof of concept it appeared that the connected module had more effect than just communication with this module. Probably the SIM architecture has to be extended with model modules containing types that are related to the connected external module.

The programming language of the proof of concept

The programming language ELM caused some differences with the architecture. One difference is the control of the view functions. As far as we could see it was only possible to

define one view function that has to control the other view functions. The update of the views is done automatically, but the views are not called from the control modules of the entities.

Another big difference are the model modules. Our architecture is based on a traditional MVC-model architecture in which the module modules are used to store values of variables and used to supply those values to functions that need them. In ELM only variables that are defined in the main module are available to functions of other modules. We have used the model modules to define the types belonging to the variables and used those in the definition of the variables in the main module. The effect of this is a lower maintainability when an optional functionality has to be deleted completely from the SIM.

9. Conclusions and recommendations

9.1. Conclusions

The goal of our study was to examine a Student Interaction Module that can be used as part of an ITS that supports stepwise Logic exercises and is based on a traditional or four model architecture, a combination of modules from different ITSs with a four module architecture or a learning environment. To do this we have examined three research questions, that are mentioned below.

RQ 1. what are the requirements for the Student Interaction Module?

We have determined the functional and non-functional quality guidelines and the most important ones of those by doing a literature study and an interview. The most important requirements and quality guidelines are:

- Compatibility; the ability to connect to different ITSs from the SIM.
- Maintainability; how easy can the SIM be maintained, because of the many possible connected ITS there are a lot of possible configurations that have to be maintained.
- Optionality; not every ITS has the same functionality, so the SIM has several optional functions.

RQ 2. what are the trade-offs with these requirements?

The most important trade-off between the requirements is compatibility versus maintainability; the SIM should be able to connect to different ITSs, but not every ITS offers the same functionality and the connection to every ITS can be different. This could lead to complex software with many specific parts for specific ITSs, but SIM also has to be maintainable, otherwise it will be too expensive to maintain and has possible more errors.

For the architecture of the SIM we have also looked at the trade-offs and concluded that a web-based, product line architecture based on an MVC-model will fit the best with the determined requirements. Web-based because this gives the possibility to make the SIM available for more students. A product-line architecture because this gives the possibility to use different modules for different connected ITSs if there is a difference in functionality between those, but it gives also the possibility to re-use modules for functionality that is the same in different ITSs. A plug-in architecture whereby other developers can add functionality to our SIM is, based on the requirements, not needed, but an extra complex solution because this kind of software has to be prepared for all kind of extensions in functionality. The MVC-model and the Flux architecture both fit our requirements. We have chosen to use the MVC-model because we have more experience with this one, but we think the Flux architecture could be used too.

To describe the architecture of the SIM we have used a view with a feature-model. This is a model that is focused on defining the fixed and the optional functionality. We also used a functional view that is focused on showing the complete functionality of the system. These combined views give a good picture of the architecture and the mandatory and optional functionalities.

To prove that the architecture can be used we have built a proof of concept based on a part of LogEx that connects with the IDEAS framework. Because this part only uses a part of the complete architecture we also described how other parts of the architecture can be implemented in the proof of concept.

The proof of concept shows that not only the communication of the SIM is related to the connected module, but the connected module effects also the type definitions of the other entities. An extra entity in the architecture related to the connected module is suggested. This module can be used by the variables of the entities in the program as a facade for the definitions of the variables related to the connected module.

RQ 3. what is the flexibility of our architecture?

Because there are many logic ITSs based on a four model architecture, It is almost impossible to prove that the architecture is suitable for each of these ITSs, furthermore new ITSs can be developed and for those it is impossible to prove that they can use the SIM architecture. Therefor we have examined two other ITSs that are non-logic ITSs and looked if those ITSs could use the SIM architecture. If those non-logic ITSs can use the SIM architecture, there is a bigger chance that other logic ITSs also can use the SIM architecture. We have described how those two ITSs can use the SIM architecture.

Conclusion

Our conclusion is that the described SIM architecture can be used for implementing a student interaction module, but the adding of an extra entity related to the connected module is advised. Besides that the relation between feedback and hints in relation to the SIM architecture needs to be studied.

9.2. Future work

The suggested extra module related to a connected module has to be studied deeper. In our study this is only suggested, but a full study of the effects of the effect of the connected module for the SIM still has to be done.

Connect another ITS to the proof of concept, to show that the architecture is still usable. Until now it is only a paper prove.

Connect two other ITSs both with the same standard communication interface like QTI, OpenMath or MathML. According to our study this should result in one set of generic communication modules that can be used for both ITSs, whereby there will not be special parts of code related to a specific one of the ITSs.

A closer study on the relation between hints and feedback related to the SIM architecture. In our architecture we have separated these entities, but in our proof of concept we have combined them and we also see that in other ITSs these seem to be combined. There are more studies on hints and feedback, those can be combined with this study on the SIM architecture to determine if another structure of feedback and hints is better.

Bibliography

Aleven, V., Baker, R., Blomberg, N., Andres, J.M., Sewall, J., Wang, Y., Popescu, O., 2017. Integrating MOOCs and Intelligent Tutoring Systems: edX, GIFT, and CTAT, in: Proceedings of the 5th Annual Generalized Intelligent Framework for Tutoring Users Symposium, Orlando, FL, USA. p. 11.

Aleven, V., McLaren, B.M., Sewall, J., Van Velsen, M., Popescu, O., Demi, S., Ringenberg, M., Koedinger, K.R., 2016. Example-tracing tutors: Intelligent tutor development for non-programmers. Int. J. Artif. Intell. Educ. 26, 224–269.

Alpert, S.R., Singley, M.K., Fairweather, P.G., 1999. Deploying intelligent tutors on the web: An architecture and an example. Int. J. Artif. Intell. Educ. 10, 183–197.

Anderson, J., 1987. Production systems, learning and tutoring., in: Klahr, D., Langley, P., Neches, R. (Eds.), Production System Models of Learning and Development. MIT Press, Cambridge, MA, USA, pp. 437–458.

Anderson, J.R., Boyle, C.F., Reiser, B.J., 1985a. Intelligent tutoring systems. Science 228, 456–462.

Anderson, J.R., Boyle, C.F., Yost, G., 1985b. The geometry tutor., in: IJCAI. pp. 1–7.

Birsan, D., 2005. On plug-ins and extensible architectures. Queue 3, 40–46.

Boduch, A., 2016. Flux architecture. Packt Publishing Ltd.

Brooks, F., Kugler, H., 1987. No silver bullet. April.

Brusilovskiy, P.L., 1994. The construction and application of student models in intelligent tutoring systems. J. Comput. Syst. Sci. Int. 32, 70–89.

Brusilovsky, P., 1995. Intelligent learning environments for programming: The case for integration and adaptation, in: Proc. of AI-ED. pp. 1–8.

Corkill, D.D., 1991. Blackboard systems. AI Expert 6, 40–47.

Dermeval, D., Leite, G., Almeida, J., Albuquerque, J., Bittencourt, I.I., Siqueira, S.W., Isotani, S., Silva, A.P.D., 2017. An ontology-driven software product line architecture for developing gamified intelligent tutoring systems. Int. J. Knowl. Learn. 12, 27–48.

Flux [WWW Document], 2019. . Flux - Appl. Archit. Build. User Interfaces. URL https://facebook.github.io/flux/ (accessed 1.23.20).

Gerdes, A., Heeren, B., Jeuring, J., van Binsbergen, L.T., 2017. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. Int. J. Artif. Intell. Educ. 27, 65–100. https://doi.org/10.1007/s40593-015-0080-x

Goguadze, G., 2010. ActiveMath-generation and reuse of interactive exercises using domain reasoners and automated tutorial strategies.

Goguadze, G., 2009. Representation for Interactive Exercises, in: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (Eds.), Intelligent Computer Mathematics. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 294–309.

Goguadze, G., Mavrikis, M., éalez Palomo, A.G., 2006. Interoperability Issues between Markup formats for Mathematical Exercises. WebALT 2006 Proc. 69.

Hartley, J., Sleeman, D.H., 1973. Towards more intelligent teaching systems. Int. J. Man-Mach. Stud.

Heeren, B., Jeuring, J., 2014. Feedback services for stepwise exercises. Sci. Comput. Program. 88, 110–129.

Ingeno, J., 2018. Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts. Packt Publishing Ltd.

ISO/IEC 25010:2011 [WWW Document], 2011. . ISO. URL https://www.iso.org/standard/35733.html (accessed 3.11.19).

Kimball, R., 1982. A self-improving tutor for symbolic integration, in: Intelligent Tutoring Systems. Academic Press.

Kohlhase, M., 2006. OMDoc–An Open Markup Format for Mathematical Documents [version 1.2]: Foreword by Alan Bundy. Springer.

Krasner, G., Pope, S., 1998. A cookbook for using the model - view controller user interface paradigm in Smalltalk - 80. J. Object-Oriented Program. - JOOP 1.

Lehman, M.M., 1980. Programs, life cycles, and laws of software evolution. Proc. IEEE 68, 1060–1076.

Lodder, J., Heeren, B., Jeuring, J., 2016. A Domain Reasoner for Propositional Logic. J. Univers. Comput. Sci. 22, 1097–1122.
Mathtutor [WWW Document], 2020. URL https://mathtutor.web.cmu.edu/home (accessed 8.18.20).
Murray, T., 1999. Authoring intelligent tutoring systems: An analysis of the state of the art.
Narciss, S., 2008. Feedback strategies for interactive learning tasks. Handb. Res. Educ. Commun. Technol. 3, 125–144.
Narciss, S., 2005. Informatives tutorielles Feedback: Ableitung und empirische Überprüfung von Entwicklungs-und Evaluationsprinzipien auf der Basis instruktionspsychologischer Erkenntnisse. Waxmann Verlag.
Nkambou, R., Bourdeau, J., Mizoguchi, R., 2010. Advances in Intelligent Tutoring Systems, Studies in Computational Intelligence. Springer, Heidelberg.
Northrop, L., Clements, P., Bachmann, F., Bergey, J., Chastek, G., Cohen, S., Donohoe, P., Jones, L., Krut, R., Little, R., others, 2007. A framework for software product line practice, version 5.0. SEI–2007–httpwww Sei Cmu Eduproductlinesindex Html.
Nwana, H.S., 1990. Intelligent tutoring systems: an overview. Artif. Intell. Rev. 4, 251–277.
O'Shea, T., 1979. A self-improving quadratic tutor. Int. J. Man-Mach. Stud. 11, 97–124.
O'Shea, T., Bornat, R., du Boulay, B., Eisenstadt, M., Page, I., 1984. Tools for creating intelligent computer tutors, in: Proc. of the International NATO Symposium on Artificial and Human Intelligence. Elsevier North-Holland, Inc., pp. 181–199.
Padayachee, I., 2002. Intelligent tutoring systems: Architecture and characteristics, in: Proceedings of the 32nd Annual SACLA Conference. Citeseer, pp. 1–8.
Patvarczki, J., Politz, J., Heffernan, N.T., 2009. Scalability and Robustness in the Domain of Web-Based Tutoring, in: Scalability Issues in AIED Workshop at the 14th International Conference on Artificial Intelligence in Education.
Ritter, S., Brusilovsky, P., Medvedeva, O., 1998. Creating More Versatile Intelligent Learning Environments with a Component-Based Architecture. pp. 554–563. https://doi.org/10.1007/3-540-68716-5_61
Ritter, S., Koedinger, K.R., 1996. An architecture for plug-in tutor agents. J. Artif. Intell. Educ. 7, 315–348.
Self, J., 1998. The defining characteristics of intelligent tutoring systems research: ITSs care, precisely.
Siemer, J., Angelides, M.C., 1998. A comprehensive method for the evaluation of complete intelligent tutoring systems. Decis. Support Syst. 22, 85–102.
Silva Marcolino, A., Francine Barbosa, E., 2017. Towards a software product line architecture to build m-learning applications for the teaching of programming, in: Proceedings of the 50th Hawaii International Conference on System Sciences.
Sosnovsky, S., Dietrich, M., Andrès, E., Goguadze, G., Winterstein, S., 2012. Math-Bridge: Adaptive platform for multilingual mathematics courses, in: European Conference on Technology Enhanced Learning. Springer, pp. 495–500.
Sottilare, R.A., Brawner, K.W., Goldberg, B.S., Holden, H.K., 2012. The generalized intelligent framework for tutoring (GIFT). Orlando FL US Army Res. Lab. Res. Eng. Dir. ARL-HRED.
Teeuwen, G., 2016. Comparing architectural styles for distributed expert knowledge modules in intelligent tutoring systems.
Thüm, T., Kästner, C., Erdweg, S., Siegmund, N., 2011. Abstract Features in Feature Modeling, in: Proceedings - 15th International Software Product Line Conference, SPLC 2011. https://doi.org/10.1109/SPLC.2011.53
VanLehn, K., 2006. The behavior of tutoring systems. Int. J. Artif. Intell. Educ. 16, 227–265.
Wenger, E., 1987. Artificial Intelligence and Tutoring Systems: Computational Approaches to the Communication of Knowledge. Morgan Kaufmann Publ.

Appendix

A. Adding other functionality

In this appendix we discuss functionality that is part of the designed architecture but is not included in our proof of concept. Per entity we roughly describe how the missing functionality can be added to our proof of concept and what functionality the external module should support.

A.1. Exercise selection

SIM selects exercise

For SIM selects exercise the external module should have a list of exercises that can be retrieved, or our proof of concept should have a hardcoded list of exercises. When a list of exercises can be retrieved from the external module, there should be a function in CommunicationControl to control the retrieval of the exercises, this function will need a SendRequestType that describes the encoded message that is send to the external module. This SendRequestType should be added to the function methodCall of CommunicationControl and maybe new encoders are needed to encode parts of the new message. Also an extra decoder is needed to decode the new messages from the external module. The new function in CommunicationControl uses the encoder and the decoder to call the existing sendRequest function of CommunicationModel. To start the retrieval of the exercises a new function in ExerciseSelectionControl is needed, this function calls the new function in CommunicationControl. The list of exercises can be added to the model (an addition to the Model type) by another new function in ExerciseSelectionControl. To trigger both new functions in ExerciseSelectionControl new messages of type Msg are needed.

A hardcoded list can be added to ExerciseSelectionModel. This list can be read by a new function in ExerciseSelectionControl and stored into a field of the variable of the Model type.

The selection of a new exercise can be done for both options in the same way. At this moment our proof of concept only holds the current exercise as part of the steps taken, it is easier to change to Model type and add an extra field for the current exercise. This extra field can be used by a new function in ExerciseSelectionControl to determine the next exercise. After an exercise is completed a new message of type Msg can be given. This will be handled by the update function of MainControl and the new function for selecting the exercise in ExerciseSelectionControl is called.

The external module selects the new exercise

For this option the external module should support the selection of new exercises. When using a stateless external module like the Ideas framework this means that the external module does not keep track of which exercise is presented to the student, thus the SIM has to send that information to the external module. When using an external module that stores this information the SIM does not have to provide this information.

The functionality that has to be added to the SIM can look like the function generateRandomExercise from CommunicationControl for an external module that holds the state and like the function retrieveExample from CommunicationControl for a stateless external module. In both cases these functions can be triggered by new messages of the type Msg, that are handled by the update function of MainControl.

User defined exercises

For user defined exercises the external module should support exercises defined by the student. In SIM there should be an input field for the user defined exercise. This exercise can be stored in the current state in the Model type. The exercise can be send to the external module with a function that looks like the checkStep function of CommunicationControl.. This can result in an error message when the exercise is not good. That message can be handled by FeedbackControl and FeedbackView functions. When the exercise is correct it can be added as a normal exercise like the exercises returned by the functions generateRandomExercise and retrieveExample from CommunicationControl using a SetState message from the type Msg.

A.2. Answer

Delete

The delete functionality deletes all steps starting with a selected step. In order to make this functionality it should be possible to select a step from the list of taken steps, this should be added to the answerView functionality in AnswerView. With a new message of type Msg it is possible to send the selected step to a new function in AnswerControl that deletes all involved steps. This new function can also be used to support the current undo functionality; undo is the deletion of the last entered step.

Redo

The redo functionality can be used with the undo and the delete functionality. The undo and delete functionality could store the steps that are deleted in an extra field with an list of deleted steps in the type Model. The redo functionality can be called by pressing a new button that uses a new message of type Msg. That message is send by the update function of MainControl to a new function in AnswerControl and that new function adds the step by using the existing addStep function from AnswerControl and afterwards it removes the added step from the list of deleted steps. The button for redo can be active as long as there are steps in the list of deleted steps.

Complete answer

Our proof of concept only supports step based exercises. An easy way to change this to exercises that are only check after completing the exercise is to change the functionality after pressing the submit button in such a way that the step is added to the steps taken without checking the step and add an extra button for checking the complete answer whereby the steps taken are send to the external module in the same way as done in the proof of concept. When one of the checks of a step result in an error, the complete exercise is answered wrongly.

Another way in which the complete answer is send to the external module depends on the possibilities of the external module. If the external module supports complete answers the current functionality after pressing the submit button can still be changed in only adding the step without checking it and adding an extra button to check the complete answer. This extra button should than call a new function in CommunicationControl that can handle a complete list of steps. For this there will be a new message of the SendRequestType needed, that can be handled by the function methodCall of CommunicationControl. Also there will be also a new encoder needed that can transform a list of steps into a message and the new message can be send with the sendRequest function to the external module. The return message can be the same as the one from the checkComplete function of

CommunicationControl and stored in the ApplyReady type that can be handled by the MessagesView functionality of MessagesView.

Answer in two directions

An answer in two directions can be used by exercises that prove logical equivalence. For this a second list of steps taken in the Model type can be used. In AnswerView this second list of taken steps should also be shown to the student. There should also be a way for the student to indicate whether the answer should be added to the top steps or the bottom steps. The current submit button can be used to indicate that the step should be added to the top steps, and an extra button can be added to indicate that the step should be added to the bottom steps.

How the checking of the steps is done depends on the possibilities of the external module. It is possible that the bottom steps are checked in the same way as the steps in the proof of concept are checked with an addition that the direction of the step also should be send to the external module. Another completely different approach is also possible, for example all the steps taken are send to the external module. Both options look like other options we have described.

Line numbers

With axiomatic logic exercises line numbers can be used to refer to lines on which a rule is applied. Line numbers can be added answerView as an input field. In AnswerModel a type for the line numbers can be defined and in AnswerControl a function to check the validity of the line number (i.e. when given, a line number must be an integer between 1 and 1000) . Rules can be applied on one or more line numbers, thus in AnswerView there also should be a possibility to add the line numbers to which a rule is applied; a list of line numbers with a type definition in AnswerModel.

Formula in parts

LogAx[13] has the possibility to add a formula in parts based on the rule that is applied. With the entity Rule we look at how the rules can be added, for Answer it should be possible to split up the formula in several parts. This can be done by adding multiple input fields for the formula in pieces and concatenate those pieces at the moment the submit button is pressed and send that complete formula to the external module with a function alike checkStep from CommunicationControl. It is also possible to send the individual pieces to the external module if that is needed. In that case the formula can be concatenated after it is successfully checked.

Case structures

LogInd[14] is used for inductive logic exercises and uses case structures for answering. Those case structures are of specific types and every type can hold one of more cases, and each case has one or more steps. This structure can be defined in the Model type. AnswerView will need some extra buttons for adding and removing cases and It will need a way to select the type of case that is used. The rules that can be used in a case depend on the case, which means that for every case that is shown the rules has to be determined (also see variable rules).

[13] http://ideas.cs.uu.nl/logax/

[14] https://ideastest.science.uu.nl/logind/

A.3. Rule

Rules as optional functionality

Rules are optional in the architecture; the Ideas framework asks for steps and formulas when solving an exercise, but other external modules do not need to have this functionality. An easy way to remove the rules from the proof of concept is to delete the part inputType = rulesSelector model in the function answerLine in AnswerView, this way the rule input for the student will not be shown. The function handleStep of AnswerControl has to be changed in a way that it does no longer check on a selected rule. The function getRuleList from RuleControl has to be changed in a way that it does not call the function retrieveRuleList from CommunicationControl, this will prevent that rules will be retrieved from the external module and the checkStep function of CommunicationControl has to be changed in a way that it does not send the rule to the external module. Problems with this solution are that hints will still show rules and there will be lots of dead code in the software. For a complete solution everything related to rules has to be deleted. This means changes in MainControl (all calls to the Rule modules has to be deleted), MainModel (the types Model and Msg have items related to rules), the Answer modules have to be changed, the Rule modules have to be deleted, the Hint modules have to be changed (hints contain rules) , the Feedback modules have to be changed (these show the rules as part of messages) and the Communication modules has to be changed (rules are part of the messages send to and received from the external module).

Hardcoded rules

In our proof of concept the available rules are determined by the external module. If the external module does not have an option to retrieve the rules, the rules can be hardcoded in the SIM. This would mean that RuleModel holds a list of rules, that can be read by a function like requestRules that returns a message AddRules with a list of rules. This list will be put in the Model type variable by handleRulesReceived.

Transformation of rules

In our proof of concept we use the rule ids from the Ideas framework. These rules can be transformed to other text fields by adding a translation function, that translates the ids into more readable text. This can be done by adding a translation table to RuleModel in which for every id from the Ideas framework a text translation is given. Problem with this solution is that all ids of the Ideas framework should be in the translation table and if something changes to a rule id in the Ideas framework, this change should also be done in the translation table.

Variable rules

Variable rules are needed when the set of available rules depends on the answer line. Variable rules are for example used in LogInd where depending on the case that is filled, the rules differ. The set of the available rules can be determined by the external module, but if the external module does not provide them, they can be determined in the SIM in a similar way as described with transformation of rules.

A.4. Hint

Hints as an optional functionality

In the architecture hints are an optional functionality. The easiest way to implement this in our proof of concept is by disabling the hint related buttons or remove those buttons from the getAllButtons function in the Buttons module. Another easy to implement option is to change the getNextHint function of HintControl and return an unchanged model with Cmd.none or change the following lines in getNextHint

```
( { model | hint = HM.RuleName, feedback = FM.HintFeedback ("Hint 1: Apply the rule " ++
hint.rule) }, Cmd.none )
```

Into

```
( { model | hint = HM.NoHint, feedback = FM.HintFeedback ("No hint available") }, Cmd.none )
```

This will cause that the text "No hint available" is shown to the student and the next hints will never be reached. Using this option will need some clear comments in the code otherwise this could lead to unclear code. Both of these options lead to dead code in the software.

A complete solution is to remove everything that is related to hints. In order to do this the next changes has to be done:

- The modules HintControl and HintModel
- The hint field in the type model
- Delete the following messages from type Msg: FetchHint, AddHint, FetchCompletion and SetCompletion.
- Delete the lines of code using the above messages from type Msg from the update function of MainControl
- Remove the following functions from CommunicationControl: retrieveCompletingSteps, retrieveHint, decodeHint, decodeSolution and decodeSolutionStep
- Remove SendOnefirsttext and SendSolution from the type SendRequestType in CommunicationModel
- Delete the types SolutionFields and OnefirsttextFields in CommunicationModel
- Remove the options SendOnefirsttext and SendSolution from the function methodCall in CommunicationControl

An example as hint

This is an option that should be supported by the external module before it can be implemented in the SIM. Implementation looks like the completion of an answer as a hint and can possibly use the same functions.

The complete solution as a hint

In our proof of concept this can be added by getting the original exercise from the list of steps taken and then retrieve the complete solution with the completion of an answer functionality.

Transforming hints

The text of the hints can be transformed in the way as mentioned with rules.

Retrieving more than one hint

In our proof of concept only one hint is retrieved from the Ideas framework and that one is copied into the second and third hint. If the external module would return all three hints at the same time this would mean a change in the Hint type (the other types of hints should have their own fields), the decoder decodeHint (should decode all the fields) and a change in the function getNextHint (the values from the new fields should be used). Another option could be that the external module returns only one hint at the time depending on the last hint given. With a stateless external module this would mean that the external module should have a way to inform it about the given hints and the SIM has to supply that information when asking for a next hint. With a stateful external module SIM only has to ask for a next hint and the external module will know which hints are already given and gives the next hint without additional information.

Separate function for applying the next step

In our proof of concept the next step will be applied after three hints are given and the student asks for a next hint. By adding an extra button and a separate function in HintControl this applying of the next step can be made separate from the giving of next hints.

A.5. Feedback

Optional feedback

In our architecture the feedback messages are optional, these are the messages for the student that are received from the external module after diagnoses of a step or a complete answer. In our proof of concept the Feedback modules also show hints and system error-messages.

To remove only the feedback messages that are results of the diagnoses the function handleFeedback has to be changed. Although this function does not give a message, it stays on the same input line when a false answer is given. This can also be seen as a notification to the user that the answer was incorrect. This has to be changed; the function should always add the answer line to the list of steps taken. The function handleReadyAction only gives a message whether the answer is complete or not. This functionality can be deleted by deleting the ready button, changing the update function in MainControl and deleting this function. This will not affect the functionality of the rest of the program.

Step feedback messages

In our proof of concept the submitted answer line is only added to the list of steps taken when the result of the diagnose is Expected, all other possible outcomes are ignored. In order to change this the function decodeDiagnose of CommunicationControl and handleFeedback of FeedbackControl have to be changed. decodeDiagnose has to have means to decode the other kind of results and handleFeedback has to be able to show these results.

Transformation of feedback

Feedback messages that come from the external module can be transformed by using a hardcoded transformation table in FeedbackModel in a same way as described with rules.

Different kinds of feedback

In our architecture we have given the possibility to use different kinds of feedback. In our proof of concept there is no difference made between the types of feedback. AUC combined with KR feedback is given after diagnosis of a step; the given answer is only added to the list of steps taken when the step was correct, in all other cases the step was incorrect but there is no EF feedback given, thus the student does not get any information about that was wrong with the step. After diagnoses of a complete answer only KR feedback is given; the answer is complete or the answer is not complete.

Because the Ideas framework we use is stateless it does not hold any information about performance of a student for a set of exercises. In order to give that KP feedback the SIM should store information about the results of every exercise. This can be done by adding a result table to the Model type in which is the result for every started exercise is stored.

KRC feedback can be added if the external module supports this and returns the right answer after giving a wrong answer. This could be shown as a message.

MTF feedback can be implemented in the proof of concept by adding a counter that holds the number of tries for a step to the Model type. When the maximum number of tries is

reached the correct step can be retrieved like is done with hints and added to the list of steps taken or the next exercise can be retrieved.

EF feedback has to be supplied by the external module; when the external module supplies that information it can be added to the program. Maybe an extra feedback message part is needed in the page that is shown to the student, this can be added in a similar way as feedback is added to it.

A.6. Communication

Communicate with other external modules

In our architecture there is a clear difference between the communication protocols, the communication actions and the communication modules. In our proof of concept the communication protocols are defined in the encoding and decoding functions, the communication actions are the functions that are available for the outside world of the CommunicationControl module checkComplete, checkStep, generateRandomExercise, retrieveCompletingSteps, retrieveExample, retrieveHint and retrieveRulelist and the communication module is defined in the function getConfig from CommunicationModel that uses the type Config that can be used when information about the connected external module is needed.

The decoding and encoding functions are part of the other communication format possibility. When the external module uses one of the other formats the decoding and encoding functions has to be changed to that format. The feedback services can be implemented like the ones we already have. The communication module will be a change of the definition in getConfig and probably the sendRequest function of CommunicationModel has to be changed because this combines the communication format with a specific communication module.