SINF22 Master Thesis

A Study Program Verificator Based On Petri Nets

By:

Colson Olivier

SINF22MS

50391000

Advisor: Lobelle Marc

UCL, 2014-2015

Table of contents

Abstract

One of the key periods of the academic year consists of the moment the students select the courses they will follow. Indeed, this choice determines the entire year of the student, and it is hence very important for it to be carefully checked before being accepted.

For this, the current system relies on the program commission members, responsible to manually check each of the proposals, and dialog with the student submitting it if anything seems unclear or wrong into it. Furthermore, the program commission is also responsible to manage "irregular" cases that could arise, for example with exchange students, as mapping needs to be done between the courses they have followed abroad and their equivalent, say, at UCL.

This document presents a web application to integrate in a pre-existing university computer infrastructure in order to ease the dialog between the students and the commission. This application also introduces an automatic study program verification tool, performed using Petri Nets structures. We precisely define here how and why this tool has been implemented, and how it eases the communication between students and the commission.

1) Introduction

We present in this document a web application aiming at verifying study programs proposed by students to their university, using a Petri Net modelling. This application has been developed as a master thesis in Informatics Sciences at Université Catholique de Louvain (UCL) during the academic year 2014-2015.

We begin in section 2 (section 1 being this introduction), by explaining precisely what the application is needed for, and defining the various terms we use in the whole document to refer to key concepts. Then, in section 3, we give more details about the way the application achieves the various goals we presented. After that, section 4 gives a more technical overview of the system, explaining which exact technologies were used at which places and for which reasons. Section 5 is the most important one, as it describes in depth the Petri Net modelling that is used by the application in order to achieve study programs verification. It also details how the results obtained by this modelling are interpreted by the program so that verification reports can be built and presented to the user. Section 6 then shows that the application can easily be extended, and hence can be developed further if need arises to add new functionalities to it. Section 7 lists and explains what these new functionalities could be, and explains why they could be interesting to implement. Some of these possibilities, as we will see, have been suggested by a real user. Finally, section 8 gives a conclusion resuming quickly what has been treated in the whole document. It also gives a list of the appendices to this document, with a description of their content. Of course, this document ends out in section 9 and 10 giving a bibliography and acknowledging some people without who this work would not have been possible.

2) Analysis of the application's needs
    1. Context and goal

Selecting his courses can be challenging for a student. Indeed, there are lots of them available, some needing the student to have already followed specific courses or acquired specific skills, some others requiring a giving set of other courses to be selected together with them. Furthermore, the student must always keep in mind that he needs, in the end, to validate a study program in order to obtain a diploma.

As it is now, all this selection mechanism is generally achieved (at least, at UCL) by letting the student look on the Internet for what is available, select from it, and then send a mail to the program commission with his choices. The commission then has the responsibility to check each of the mail sent by the students, and either accepted the proposed courses list or ask the student to correct or complete it in some way.

From the preceding description, it is pretty clear that something can be done here to get this treatment more "automatic". However, one important thing to notice is that it cannot be fully automatized. Indeed, the actual way course selection and program validation is achieved is much less "formal" and "simple" than described above, as students can have followed courses outside their current university, and hence have the skills required for some courses despite the fact that it is "formally" registered into the system. These exceptions are quite numerous, and are all managed by a kind of "negotiation" that takes place between the student and the commission: the student proposes a list of courses and a justification saying that he has acquired these skills somewhere else, and the commission then decides whether or not his justification is acceptable, and validates or not his courses program. These exchanges forbid us to develop a completely automatic verification algorithm, and hence make manifest the need for a human intervention in the program verification process.

This is the spirit in which the application presented here has been developed: to give an automatic program verification tool reporting exceptions when they occur and simply telling that nothing is wrong in the regular cases.

The application also uses these reports to ease the communication process between the student and the commission. Indeed, as it is, in most cases the student will not know in advance that there is something wrong in his program, and will thus not give any justification for it, forcing the commission to send him a mail asking him to do so. Of course, it would have been easier if the student's initial program proposal had been fully justified the first time it was sent. The application hence makes it mandatory. Every abnormality found by the verification algorithm is reported to the student for justification. Only once each of the found abnormalities has been justified can the student send the proposal to the commission. This philosophy saves time for the commission, as it gives it a more global view of the student's situation at once, and does not require any further e-mail in most of the cases.

2. Terminology

In order to treat the subject in the clearest manner possible, a few terms need to be defined. This section gathers each of these, with an explanation of the concepts it covers, and the way it used in the application's model. It also provides, for each case, an example at UCL, so that the link between "theory" and "reality" is made clear. Of course, these terms are used in all the rest of this document, as well as in the application itself and its documentation.

Note that it is important we "invent" most of those terms, as using pre-existing ones could induce some conflict with their legal signification. An example of this, with the Marcourt decree, is the notion of "precondition" we define later in this section. One could refer to what we call preconditions as prerequisites, which he might find more explicit. The fact is, however, that the Marcourt decree defines prerequisites as being *courses*, and not *achievements* as it is the case here. Marcourt's prerequisites can be modeled with preconditions, but not the opposite. Hence, as our model is more general, it is better to choose it a brand new name, so that we avoid any confusion.

a. Time division

As everyone knows, each study system is based on a division of time. Our system hence makes no exception to that rule and defines several levels of time divisions. Each of them is presented in the following points. These divisions have been made so in order to match to as many different study systems as possible.

i. Choice period

The choice period is the largest time division. It corresponds to the moment when the first course selection occurs. Being the largest division also implies that it is at the end of a choice period that a course (and by extension, a program) can be validated (see the following points for a complete description of what a program or a course is).

The beginning of the choice period corresponds to the moment when the students must give the list of courses they will follow during each study period of it (study periods are defined in the following point).

At UCL, a choice period corresponds to an academic year. Indeed, it is at the beginning of the academic year that each student selects the courses he is going to follow during the entire year. This list of courses can be modified later on (as we will discuss), but a *first selection* must be made at the beginning of the year, which is exactly the definition we gave of a choice period earlier, "the moment when the first course selection occurs".

ii. Study period

Each choice period contains at least one study period. The number of study periods per choice period is fixed for all choice periods.

A study period basically is a time slot during which a set of courses both begin and end. The courses within a study period, as explained with the choice period concept, are first selected at

the beginning of the choice period containing it. However, this does not mean that study periods are immutable. Indeed, each study period's course list can be modified after the beginning of the choice period containing it (as we will see, some limitations may apply, but they depend on the deployment of the application, and can be customized at will). This is done so that the students can change their mind on the courses they select.

At UCL, study periods correspond to quadrimesters.

iii. Tick

Ticks are the smallest time division.

As for the study periods regarding to choice periods, ticks are what study periods are composed of. A study period always contain at least one tick. Again, this number is fixed for all study periods.

During a tick, some courses may begin or end. It is hence possible for a course to both begin and end during the same tick.

At UCL, ticks correspond to weeks.

b. Courses modelling

Now that we have reviewed the time division concepts, it is time to turn our attention toward the courses. Again, we illustrate each of these with a concrete example at UCL.

i. Course

As expected, the course concept is the most central one when it comes to courses modelling.

A course can be defined as a learning activity taking place from one tick to another, possibly the same, (respectively called its start and end tick) of the same study period, allowing students to acquire at least one achievement (the notion of achievement is explained in the following point). Each course also has zero or more preconditions, and zero or more co-courses (these two notions are explained later on in this section).

A course is said to be "completed" when a student satisfying all of its preconditions and (see later for what this precisely means) has followed the whole course (hence, as soon as its end tick is over).

On the other hand, a "validated" course is a course that was completed is some previous choice period and for which the student passed the exam, hence validated the corresponding credit.

Courses are all associated to a unique identifier (called "tag"), in the form of a String, as well as a name.

A course at UCL is for example the informatics' introduction for SINF students: "LSINF1160" (this is its identifier, or tag), "Introduction à l'algorithmique et à la programmation" (this is its

name). We will keep this example to illustrate the following points, so that we give a complete explanation on how this course is integrated by the modelling explained here.

ii. Achievement

The "achievement" concept used here is inspired from the video games world, in which an achievement is basically a reward you unlock after performing some actions.

An achievement is basically a skill that is obtained by following a course from its beginning to its end. The set of assignment related to a given course hence represents the whole knowledge available by following this course. Each achievement is represented by a unique label. To make the link with the Marcourt decree, achievements are what should be used to model the "learning outcomes" of a course.

We say an achievement is "acquired" if one of the courses providing it has been completed or validated. It is hence possible for an achievement to be acquired several times.

As an example, the following list of achievement could be given to LSINF1160: Java, Object-oriented programming, Specifications basics.

From the previous description, the attentive reader might have noticed that the way the matter of a course can be divided between achievements is by far not unique. There are lots of possibilities to define achievements for a course, some finer-grained than others. This is discussed in more details later in this document, and actually depends on the deployment of the application.

iii. Notion of precondition

A precondition is a requirement on one achievement telling that it must have been completed in order for a given course to be followed by a student. Hence, it is impossible without an intervention of the program commission for a student to follow a course from which he does not have validated all the preconditions.

We call "validated" a precondition whose related achievement have been acquired.

Ton go on with our LSINF1160 example, LSINF1160 would not have any preconditions (as it is an introductory course given in first year). However, LSINF1161, LSINF1160's sequel, would have all of LSINF1160's achievements as preconditions.

iv. Notion of co-courses

A co-course x of a course y is a course that must be followed during the same choice period than y. Hence, if a student selects y, he must also select x.

Note that the co-course link is not bidirectional. Indeed, "x is a co-course of y" this does not mean "y is a co-course of x", as one course may react in real time to what is seen in the other, while the other is perfectly fine as a stand-alone course.

The notion of co-course can represent various kinds of dependencies. As we mentioned in the previous paragraph, it may simply correspond to the fact a course uses the matter of another at the same time it is explained in that course, but it can also model the fact that the content of a course changes each year and hence needs to be followed with a set of other courses of the same nature during the same choice period, in order to give a complete view of the subject (this can be true with courses depending highly on the news, in social-oriented formations, for example).

Back to our example, LSINF1160 could be set as a co-course of LSINF1161, as LSINF1161's purpose is to complete what has been seen in LSINF1160. Hence, the co-course link would here model the fact that LSINF1161 may be used to complete LSINF1160 if the teacher ran out of time at the end of that course, for example. As it is very unlikely that the teacher lacks the same amount of time, and thus always stops the matter at the same point, it is better to follow the two courses during the same year, hence the co-course link.

c. Study program

Not to be confused with a "program" (see next point), a study program is simply a set of courses selected by a student.

d. Program

A program corresponds to a diploma that can be delivered (if the student passes all his exams) once a given (of course, non-empty) set of achievements have been acquired. A program is always associated to a unique String identifier.

Examples at UCL include SINF11BA, SINF22MS, INGE11BA …

Each of these programs also is associated to a list of other programs (possibly empty), corresponding to the other programs that must be completed (see below for what this means) in order to complete this program.

A typical example of this last point is that it is impossible to receive the validation of the second year of the bachelor without having completed the first one. Hence, SINF11BA would be the only element in the required programs of SINF12BA.

As always, we will designate as "completed" a program whose achievements have all been acquired and required programs validated.

3. Assertions

Let us now briefly explain the few assertions that were made upon this model, and why they were made.

a. Courses on multiple study periods

As we have seen, the model considers it is impossible for a course to spread on multiple study periods. This choice has been made in order to keep the model as simple as possible, and clarify the structure of the Petri Net used for verification (this is discussed later).

Despite the fact it can seem a limitation, it is not. Indeed, to make such a course, one simply needs to create one course per study period, and put all these courses as co-courses one of the other, so that they need to be taken together during the same choice period.

b. Preconditions and end tick of courses

We have already mentioned that a course got completed at the end of its end tick, if its preconditions were satisfied. This has an important impact on preconditions. Indeed, any achievement acquired thanks to the course's completion will have to wait until the end of the course's end tick to be granted (except if these achievements are obtained from other courses, but let us consider that this is not the case here). This leads to the fact that it is impossible for another course to take any of these achievements as a precondition before the end of the other course's end tick, hence, before the tick following that end tick.

Put otherwise, this means that no course y needing some course x to be over can be taken if it begins in the tick course x finishes in.

This point is important to keep in mind, as it can be a bit counterintuitive.

3) What the application allows
    1. A small part of a bigger world

Before talking in more details about the functionalities offered by the system, it is important to briefly mention a point we will treat in more details in further sections: the application is to be used in a greater environment, to which it must be adapted via the implementation and extension of some specific parts of the system (detailed in section 4). Put in a less "conceptual" way, this means the application has been built, as we mentioned in the abstract of this document, in order to be integrated to an existing courses and studies management system.

This point is essential, as it implies that some data used by the system have to come from "the outside". An example of this is user credentials management, as the database of users to which the application will be connected must be managed by some external entity, and is hence considered here as an external system the application connects to. Same goes for courses and programs data. The exact way the application is connected to these "external systems" is explained in section 4.

As some data are acquired from external systems, we don't care how they were generated, and hence assume they exist "by themselves" in all of our explanations.

    2. A dialog between two poles: students and commission

The typical study program submission process consists of a dialog between two distinct entities: the student proposing a study program, and the commission, responsible to accept it or not.

This dialog always begins by a study program proposal issued by the student. This proposal is then verified by the system. Once the verification has been completed, it is possible that certain abnormalities have been found in the program (corresponding to missing preconditions or co-courses). These can merely be mistakes or exceptions due to some particularity in the student's profile (for example, an exchange student might lack some precondition for a given course, but have followed some course in his home country giving him equivalent skills, thus allowing him to "ignore" this precondition in our formal model). These abnormalities must hence all be justified by the student before transmitting his study program proposal. Indeed, in case they are mistakes, it is more efficient that the student directly corrects them without wasting the commission's time, and if they form some exception, it is important that the student explains the situation so that the commission can choose to ignore the abnormality and accept the proposal. Once the student has justified each of the abnormalities found by the system, his study program proposal is sent to the commission.

The commission then has the possibility to accept or not each of the justifications given by the student to the abnormalities of his proposal. It must also give a justification to each of these approvals and refusals, so that the system better keeps track of the decisions made. Once this is done, the commission can either accept the study program proposal (if it has accepted all the reported abnormalities) or choose to send the proposal back to the student if it is not satisfying, either because some error justifications were no acceptable, or for some other

reason (for example, if the proposed study program does not contain enough credits, or contains too many).

If the commission accepted the proposal, the student receives a notification giving him this information. Otherwise, he receives a message asking him to either justify more accurately the abnormalities refused by the commission or correct his study program so that these mistakes don't occur anymore. Whatever his choices are, the resulting study program is then once again sent to the commission, and the dialog continues.

Once his study program has been accepted by the commission, the student may have the right to edit it during a certain period of time (depending on the deployment of the application, as we will talk about in section 4). During that period, the student can access his currently accepted program and propose modifications of it to the commission, following the dialog pattern we just defined.

3. The tickets system

In the previous point, we explained the way a student communicates with the commission in order to choose a study program. We mentioned several times that messages or notifications were "sent" without explaining any further how all this actually worked. This point addresses this question.

The application manages flows of messages with what we call "tickets". A ticket basically consists of a list of successive messages constituting the whole dialog between two users on a given subject. This concept of ticket has been inspired by the OTRS framework ([13]). We refer to each of the messages associated to a ticket under the name of "ticket part". In the case of study program proposals, the study program submission dialog with the commission for a given student corresponds to one ticket, as proposals and answers are both sent using dedicated ticket parts.

A ticket always has a creator and an owner to which it is addressed. The creator is always a single user, but the owner of the ticket may be a complete user group. In that case, each member of that group will be able to see and answer to the ticket. The members of the owner group can also choose to lock the ticket for themselves, in which case they become the single owner of the ticket. This can be particularly useful in situations where a team of people is responsible of managing some task, splitting the workload between its members so that each of them is responsible for a complete "dialog". This has for example been done with OTRS in some customer services, in which a complete mail conversation with a customer is associated to a ticket. Any member of the customer service can lock a ticket for himself, so that he becomes the only one to communicate with a given customer.

In our application, user types define user groups. All students belong to the STUDENT group, as every user responsible to accept or reject study program proposals belong to the COMMISSION group. When a study program proposal is first sent, it creates a new ticket, having the proposing user as its creator, and the COMMISSION group as its owner. The commission members then can lock the tickets if they wish to, before treating their content.

4) Technical aspects

This section explains in more details which technologies were chosen to create the application and why. Then, its second part contains a brief description of the application's architecture, giving the basic keys to understand its design.

1. Chosen technologies

Let us first explain the different choices that had to be done in order to produce the application.

a. Language

The application has been developed in Java EE (JEE) 7, using Java 8 to compile the .java files.

The choice of Java EE was not that easy to make, since I did not have any experience with it and, and the wide range of different possible technologies available in it makes it very difficult to learn for beginners. I then had the opportunity, on M. Lobelle's advice, to discuss about it with M. Mulemangabo and received some clue from him as where to start and what the various acronyms of the JEE world meant.

After some more time spent learning it, I decided to choose it, because it is widely used, has a vast community of programmers, and has been available for a bit more than 15 years, which makes it very complete and well documented.

As for the choice of the version to use, the bleeding edge one seemed pretty logically the best. Furthermore, it allowed to use Java 8 (available since now a bit more than one year), whose functional programming new APIs are particularly nice to use.

b. Server

To run a Java EE website, one needs to use what is called a servlet server.

Various servlet servers exist for Java EE, each of them implementing the basic Java EE specificatoin edited by Oracle and some of them adding their own extensions to it. As some servers offer more APIs than others, the choice of the servlet server is crucial, since an application developed on one given server may not be compatible with some other. This is why the choice was made here to rely on the most standard of the servlet servers, directly provided by Oracle, and hence implementing the whole Java EE specification and nothing more: Glassfish. As for the language choice, the latest version was chosen here: Glassfish 4.

c. View Technology

Java EE offers various ways of dealing with pages display, as it seems pretty logical from a framework kept in constant evolution in the last 15 years.

One of the latest of these view technologies is called JSF. It is now quite popular since it is much higher-level than the previous view systems used by Java EE (for example, before JSF,

some explicit manipulation of HTTP request was needed in the code, this is now no longer necessary).

What JSF basically does is linking each page made on the website with one or more objects responsible of controlling the view (called “backing bean”, a “bean” being an object whose lifecycle is entirely managed by the application server, hence not instantiated by the programmer). This makes the application much more look like a classical object-oriented graphical application. Hence, Java EE with JSF is easier to understand and learn for “classical Java” (JSE) developers, and tends to ease the use of an MVC (Model, View, Controller) design, which is good as it structures the program in a quite universal and clean way.

d. Graphical library

As we briefly explained in the previous point, JSF is a view technology giving a much more object-oriented and MVC orientation to Java EE programming. However, despite the fact JSF also gives access to some very useful prefabricated graphical objects, it still requires the programmer to do a lot of interface design all by his own in order to get something fancy to show to its end user. Furthermore, JSF’s javascript support, even if it exists, still requires the programmer to do a lot by himself, even for very standard things.

To alleviate these difficulties, there exist lots of graphical libraries developed by third parties. These libraries basically provide the programmer with new HTML objects they can import and manipulate with JSF, with pre-programmed CSS and Javascript. The task of the programmer then becomes much easier, as he only needs to manipulate these objects, no more to program them.

PrimeFaces ([22]) is currently one of the most popular of these libraries. This is why it was chosen here.

e. Test database

We already mentioned that the application was to be used in a bigger environment, and some data had to be provided by independent systems. Hence, in order to test our program, it was necessary to provide some test database simulating these.

The choice for this database is to use PostgreSQL ([29]), simply because it is well known to be very efficient and is widely used.

Java EE offers several ways to connect to a database. Again, higher level solution seemed better, leading to the choice of JPA.

To make a long story short, JPA (stands for “Java Persistency API”) is a part of the Java EE specification defining an ORM (Object Relational Mapper), that is, a framework able to save objects in a database in a transparent way, building and managing the database all by its own and completely masking its functioning to the programmer.

As with servlet servers, JPA is only a specification, and various implementations of it exist. The one we selected here is Eclipselink ([21]). It is very popular in the Java EE world, and was hence preferred over the others because of the size of its community.

Note that the test database should **not** be used as-is in real deployment. Indeed, it stores all user credentials in an **unencrypted** way. This is fine for testing, but any deployment of the system should rely on some additional security library in order to at least encrypt the passwords contained in the database.

f. Petri Nets library

As we will see in the following section, and can be easily guessed from this document's title, the study program verification algorithm uses Petri Nets. We will recall what Petri Nets are later on in this document, but what is important to know at this point is that they are some kind of modelling structures allowing to be executed in order to make their "content" evolve.

The fact is Petri Nets are nearly always used together with a graphical display of their content. Indeed, for research tools or didactic use, it is far enough for the user to just look at the Petri Net to get the result he wants to. However, in our case, it is totally unacceptable to ask users to understand this modelling, because this is not "user friendly", and users basically do not care how the verification was made or what Petri Nets are. They just want a clear display of what may be wrong in the study program they are verifying, which Petri Nets are clearly not.

Hence, it was necessary to get some Petri Net library (preferably in Java, to ease its integration) that did not impose the use of a graphical display. This seemed very natural and trivial, but proved simply impossible to find on the Internet. There exist a huge variety of Petri Net libraries ([27]), but all of them seem to mix the model itself and its graphical display, hence not suiting our needs at all.

The choice was hence made here to develop a brand new Petri Nets library, with no graphical display at all: BlindPN. This library was built as a second project, and then imported into the application, as its goal is far more general than simply being used in our application. Of course, it has been programmed in Java 8.

g. Development philosophy

Nowadays, a vast majority of programmers use Integrated Development Environments (IDE) to program, like Eclipse or Netbeans. Some of them even argue that it is now impossible to make big programs without them, especially in Java. The fact is I don't agree with them and prefer to code everything using a text editor with syntactic coloration (jEdit), because my opinion is that it gives better understanding and control of the code.

The application presented here and BlindPN were hence entirely developed using a text editor and a terminal to build the project. BlindPN was compiled using the classical javac compiler, as the web application made use of Maven ([25]), a project builder widely used in coordination with Java EE (even by IDEs).

2. Architecture

We present now the structure of the application and review different interesting points in its design. Of course, this is only an overview, and a more complete description of the system can be obtained by having a look at its Javadoc. A dedicated section in the appendices explains how to generate it from the source code.

a. Drivers

We already said that the application sometimes needs some data "from the outside", and that it has been designed to be integrated in some bigger system able to provide these data.

To reflect this and make the program easier to integrate in any system, the choice has been made to define several abstract classes, called *drivers*, each of them responsible of accessing one category of data into these external systems. When the application needs some data, it just has to call the right driver for this kind of data.

The idea doing so is that, when need arises to deploy the application in some new system, no change will need to be performed into its code, but the programmer will only need to provide new implementations of these drivers, compatible with the system he wishes to integrate the application into.

To make this possible, it is of course necessary to have a way to tell the application which driver implementations to instantiate. To do so, a conf.properties file is provided together with the application, in its "resources" folder (hence in coursesCoordinator/resources, in the provided archive file). This file is parsed by the application when it is deployed in order to retrieve the names of the driver implementations to instantiate, as well as the arguments to give to their constructors if there are some.

As an example, the conf.properties file connecting the application to the test database through test drivers consists of the following text:

```
authenticationDriver=coursescoord.authentication.drivers.TestDatabaseAuthenticationDriver==CoursesCoordinator-JPA
ticketsManagementDriver=coursescoord.tickets.drivers.TestDatabaseTicketsManagementDriver==CoursesCoordinator-JPA
coursesDataDriver=coursescoord.courses.drivers.TestDatabaseCoursesDataDriver==CoursesCoordinator-JPA
programsDataDriver=coursescoord.programs.drivers.TestDatabaseProgramsDataDriver==CoursesCoordinator-JPA
```

The structure of each line of this file is the following:

driverName=driverImplementationFullName==args[0];;args[1];; … ;;args[args.length-1]

Note that the use of == and ;; as separators in this file is purely arbitrary, and is defined in the coursescoord.control.ConfigurationConstants class. Also note that the simple = separating the driver name from its implementation's information is not defined in that class, as it is imposed by the Java specification of the .properties file.

conf.properties is actually parsed when creating the coursescoord.control.DriversController class, responsible to manage access to drivers. As it is a .property file, only the drivers fields are then accessed to create drivers. Hence, if some other data were to be transmitted to the

application, they could be added into that conf.properties without any problem, as long as they respect the .properties file format (that is, as list of key-values pairs separated by the '=' character).

As we can see in the previous example of conf.properties file, four drivers have been defined:

- Authentication Driver (*coursescoord.authentication.drivers.AuthenticationDriver*)

This driver is responsible for checking the credentials of users and allowing the login process.

- Tickets Management Driver (*coursescoord.tickets.drivers.TicketsManagementDriver*)

This driver controls the way tickets and ticket parts are accessed to and saved by the application.

- Courses Data Driver (*coursescoord.courses.drivers.CoursesDataDriver*)

The courses data driver controls all the data related to the courses, the study and choice period and courses currently selected or previously validated by a given student.

- Programs Data Driver (*coursescoord.programs.drivers.ProgramsDataDriver*)

As suggested by its name, this driver is responsible for accessing the data related to programs.

Note that the driver classes' names always end with the "driver" word, a good practice that should be kept when extending the system, as it makes it easier to understand (since the general use of these classes can be directly inferred from their name).

b. Tickets modelling

We already reviewed the use of tickets and ticket parts in the application, but did not explain how this had been modeled programmatically. Let us now spend some time on it.

Tickets are modeled with the *coursescoord.tickets.Ticket* class. The Ticket objects contain all the information related to a given ticket, except the list of ticket parts associated to it (for memory savings reasons).

Ticket parts are modeled with the *coursescoord.tickets.TicketPart* abstract class. Using an abstract class here is a very key concept in understanding the portability of the application. Indeed, in the original OTRS application (from which the "ticket" concept was taken), ticket parts are merely text messages. Our application makes them study program proposals in the situations we have mentioned so far, but could we not imagine integrating a more "mail-like" system into the application, so that a student can contact the commission without having to send it his entire program? Could we not imagine sending other data than pure text or study programs? The problem is here not what is "strictly needed" by the application, but what *could* become necessary in the future. The TicketPart abstract class asks its subclasses to give

the template of the page to include when displaying the ticket part. This allows the system to dynamically load the right view when displaying any kind of ticket part, hence we only use variables of the TicketPart type, and do not care about their actual type. Hence, to add new ticket part types to the application, one simply would have to provide a tickets management driver able to load these ticket parts, a TicketPart implementation of the new type he wants to create, and a template page for displaying its content and possibly allow some answer. Of course, adding something in the system to generate ticket parts of the new type would also be useful, except if they are directly injected in the system the driver is connected to. Hence, nothing would have to change on tickets to make them able to support the new TicketPart, and no change would have to be performed in the pages displaying tickets and ticket parts, as all would dynamically change.

As it is, the system uses only one TicketPart implementation, to transmit study programs: *coursescoord.tickets. CoursesProgramTicketPart*. However, it also provides another implementation of it for ticket parts containing only text: *coursescoord.tickets.TextOnlyTicketPart*. This latter implementation was initially programmed before the former, for test purposes. It was however kept into the program, as it is fully functional, and could be integrated without any trouble in future versions of it.

c. Controllers

As we already mentioned in the preceding points, JSF uses what it calls "backing beans", that is, objects having each the task to control an entire web page or sub-page.

In the systems, each of the backing bean classes' names ends with "Controller".

d. Persistence utilities

The modelling of the system has been made in such a way that it can easily be used with an ORM framework like Eclipselink or any other JPA implementation, as in the proposed test database. Indeed, each of the objects needing to persist in some way in the system is associated with a unique identifier field (most of the time a long, sometimes a String value), which is needed by these frameworks. Of course, this does not mean it is *necessary* to use an ORM to make these classes persist. This identifier field is just present to make it possible, and give an easy way to recognize objects.

In the same idea, some class names in the application end with the "PersistenceUtil" suffix. This indicates that these classes are objects made to make it easier saving some objects into the database in the case we use an ORM pattern. Indeed, in several cases, an object must be associated with some other information in order to be saved. In that case, a PersistenceUtil object is saved, containing a reference to the original object, and some additional fields corresponding to that information.

A last point to mention about persistency concerns the way it is achieved for the test database. Indeed, most recent version of JPA allow directly putting annotations into the source code in order to express the way objects should persist. This is not what we do here, for a simple reason: the fact we use JPA is only due to the fact it was convenient for the tests. There is

absolutely no necessity to use it. As we have seen, it all depends on the drivers' implementation. Hence, this made absolutely no sense to directly put information about the way objects were stored into the objects themselves. This is why the choice was made here to rely on the way older JPA versions managed the link with the database: using an XML mapping file. This file contains basically what annotations would have contained, but is external to the source code, keeping a clear separation between the model and the way it persists (thanks to the drivers). This file can be found in the project directory (*coursesCoordinator*), at *classPathResources/META-INF/mapping.xml*. The rest of the persistence parametrization used to connect to the test database can be found in *classPathResources/META-INF/persistence.xml*.

5) Petri Nets modelling

The application uses Petri Nets to check the study programs selected by students. These are the most standard Petri Nets possible, only places containing tokens and transitions propagating them.

In this section, we explain the details of the Petri Net modelling used for the verification of study programs in the application.

1. Petri Nets

Let us begin with a small explanation about what Petri Nets are.

Petri Nets are structures composed of two kinds of entities, in any number: *places* and *transitions*.

A *place* can be seen a kind of box able to contain a non-negative number of what is called *tokens*.

A *transition* is something able to consume and create *tokens*. It can be seen as a kind of mathematical function, taking in input a set of *places*, each associated with a given value, and another set of *places* in output, each associated with a certain value too. When each of the *places* in input contains at least the same amount of *tokens* as the value the *transition* associates it with, the transition becomes able to be *fired* (or *executed*). Firing a transition consists in retrieving from all the input *places* as many tokens as the value they are associated with, and adding their associated value to the token count of all the output *places*.

Executing a Petri Net hence consists in firing one by one in random order all the transitions that are able to be executed into it. The execution finishes when no more such transition is available. Depending on the net, it is thus possible that the execution never finishes.

This brief description outlines the functioning of vanilla Petri Nets. Lots of variants exist, but we don't need them here, and so will not explain them.

Let us end with some comment about the way Petri Nets are represented in this document. We make the choice here to represent **places** as **circles containing their token count if it is superior to zero, or nothing if it is null**. We also represent **transitions** as **squares**. An **arrow from a place to a transition** means this place is one of the transition's **inputs**. Conversely, an **arrow from a transition to a place** means this place is part of the transition's **outputs**. An arrow with **no number** marked means an **associated value of 1**; otherwise, the associated value corresponds to the arrow's marking. Note that we only use associated values of 1 in our model.

2. General idea

As verification is always performed when courses are selected, the verification Petri Net corresponds to a choice period.

It is divided in several zones, corresponding to different parts of the task, which are executed one after the other. This verification is triggered by the passage of a token in the Petri Net at particular key points. This token is called the Control Token (CT). Each zone of the Petri Net will be explained in details in the rest of this section.

The execution of the net is not the verification itself. The net actually performs a treatment of the data, by propagating the tokens in it as much as possible. Then the fully executed net is processed by the application, in order to retrieve information about possible abnormalities in the study program, by checking which places contain tokens, and the number of tokens in some places.

3. Divisions of the net

As we have said, the Petri Net used for verification is divided in several zones. We now explain what each of them is used for, and how it is connected to other zones.

a. Introductory example

In order to better explain the modelling, we will refer to a common example in this whole point.

This example corresponds to a choice period of a first-year student (hence, he has followed no courses before the ones that are contained in this period). It contains two study periods, respectively named period1 and period2, each of these study periods contains 2 ticks. The example contains two courses. The first course is named course1, has no precondition, offers two achievements, named ac1 and ac2, and has the course called course2 as its co-course. The second course is called course2, takes achievement ac2 as its precondition, gives achievements ac1 and ac3, and takes course1 as its co-course (hence, the co-course link between the two courses is here bidirectional). The example also contains two programs, needing respectively ac3 and ac4 to be completed (with no program requirement).

As already mentioned, the Petri Nets presented in this whole document use common representation conventions, using circles for places and squares for transitions. The colors on the following figure have only been added for more clarity.

Places and transitions in black correspond to control, that is, elements used to segment the Petri Net into several verification zones, and mark the time division into it. Red is used for all the places and transitions used to model course1, as blue is used for course2. Green is used to mark what is related to the achievements. The arrows in orange mark the path followed by the control token when executing the Petri Net.

The green frames have been added to the Petri Net in order to mark more clearly the different zones of the net. Each of these is fully explained in what follows, as well as the way courses, achievements, and co-courses parts are built and connect to the net.

The description of each zone mentions various entities in the Petri Net, such as “course” and “co-course”. These refer to the modelling as Petri Net parts of the corresponding concepts, defined earlier in this document.

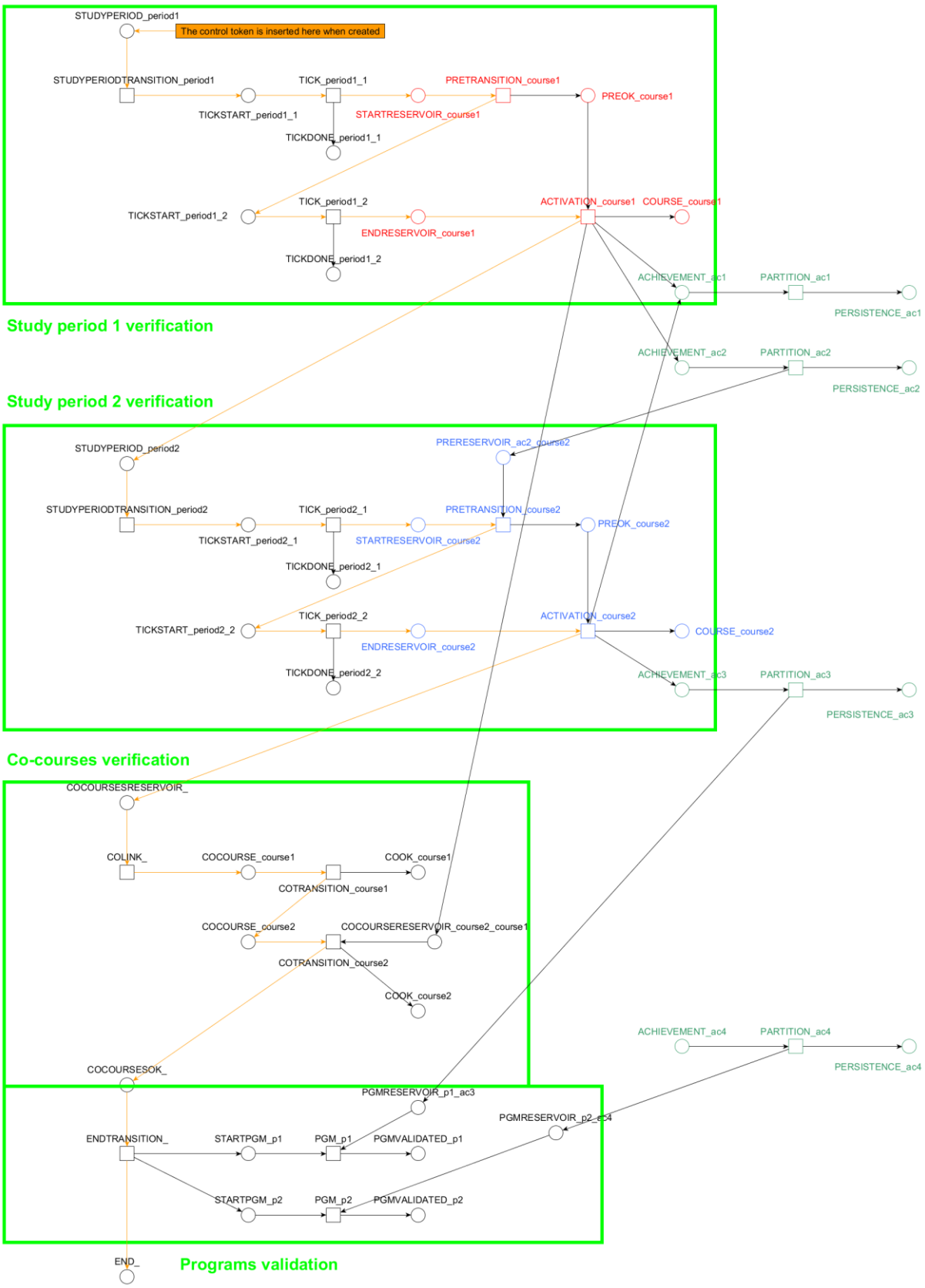

Figure 1: General structure of the Petri Net

(The names on this figure have been written as big as possible, for readability purposes. The original file can be found in the appendices' archive, to ease the reading)

b. Preconditions verification

The first-executed zone of the verification corresponds to preconditions verification.

This zone is made of a succession of subzones, each related to a study period. Hence, in figure 1, the preconditions verification zone regroups the green frames of study period 1 and 2 verification.

Precondition verification is achieved in a quite intuitive way. The control token is here used to simulate the flow of time, checking that the preconditions of each course have been validated at the time the course begins, and setting as acquired the achievements provided by a course when it is over.

The control token hence goes through each of the study period, and in each of them, successively traverses all the ticks it contains. In each of these ticks, it checks that the preconditions of all the courses beginning there can be validated, and then sets as granted all the achievements granted by the courses ending there.

The idea is here that, when a course's preconditions can't be validated (i.e. the study program contains some abnormality) the control token will get blocked. As we will see later on in this document, the verification process then just has to check where exactly the control token was interrupted in order to determine what the problem exactly is.

c. Co-courses verification

Co-course verification is the second part of the Petri Net's execution. The basic principle behind it is the same than for preconditions verification: the control token flows across the various co-courses constraints of the selected courses (however, without any notion of time or specific order here). Again, if the study program contains some abnormality, the control token gets blocked before the corresponding co-course. The precise nature of this abnormality can hence be inferred from the fully-executed Petri Net.

d. Programs completion

The last zone of the Petri Net is actually not really part of a "verification", and is hence named a "validation zone".

This zone's role is to check which program is completed by the student with the achievements he has acquired, before and during this choice period. This is done branching on the firing of the ENDTRANSITION_ transition (that is, the transition linking the end of the co-courses verification to the END_ place, which we are going to discuss in the following point), so that we try to complete each program that is *available* to the student. A program is here said to be "available" to a student if the other programs previously followed by that student fit the programs requirements of that program.

Each of the available programs is hence connected to ENDTRANSITION_ and each of the achievement it needs. A token inside the right place of a program (see later for the details)

means that this programs has been completed, which can be reported to the user after processing the fully-executed Petri Net.

e. END_ place

This place marks the end of the verification of a correct program. If the control token makes its way to it through the net, i.e. END_ contains a token after fully executing all the Petri Net, this means that the verification encountered no abnormality in the study program, neither in the preconditions, nor the co-courses.

4. Modelling of each entity

In this section, we present the details of the Petri Net modelling used by the application. Each of the following sub points addresses this question for one particular entity among the ones we defined earlier.

Each of the figures shown here represents a subpart of the Petri Net and is commented (in green), in order to provide information more directly and clearly. Red circles inside the figures correspond to entities explained in other points.

a. Choice period

As we have already mentioned, the modelling used for the verification represents an entire choice period. Figure 2 gives the global structure a whole verification Petri Net.

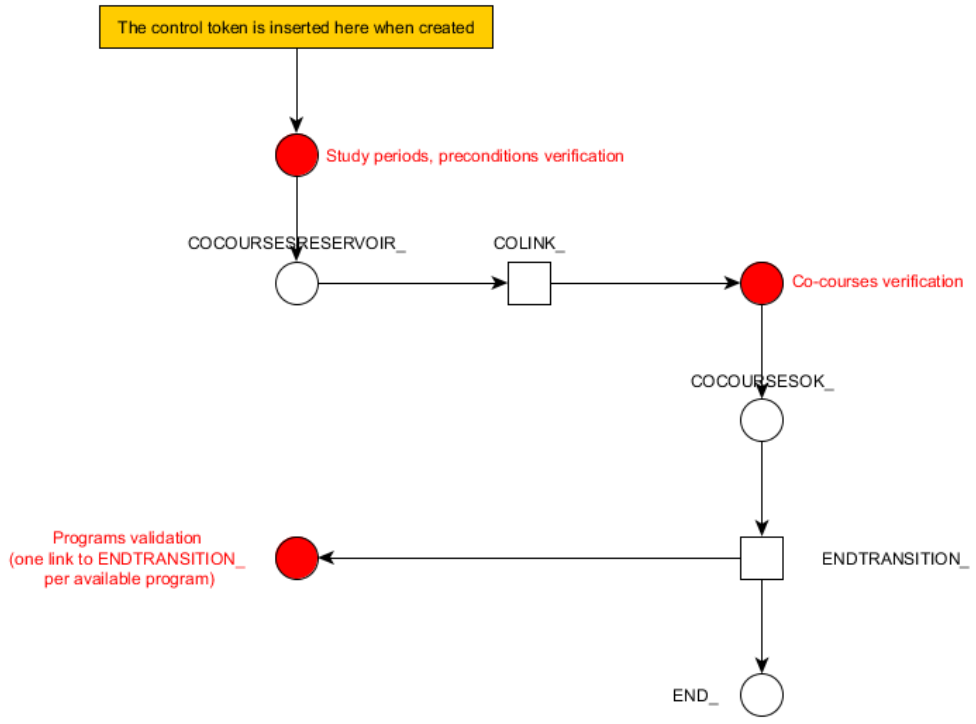

Figure 2: Petri Net modelling of a choice period

b. Study period

Figure 3 shows how a study period is modelled in the Petri Net. “p” is here to be replaced by the name of the study period concerned.

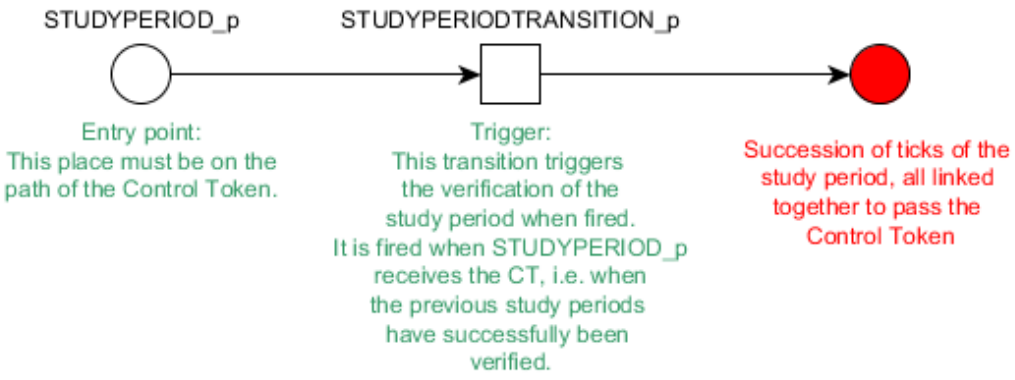

Figure 3: Petri Net modelling of a study period

c. Tick

Figure 4 presents the modelling of a tick. “p” is to be replaced by the name of the study period the tick belongs to, and “n” by the number of the tick (beginning at 1, remember that there is always at least one tick per study period).

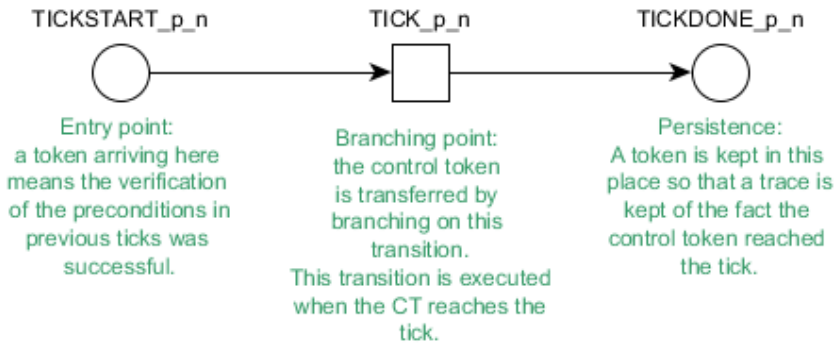

Figure 4: Petri Net modelling of a tick

d. Achievements

Figure 5 illustrates the way the concept of achievement is transposed into Petri Nets. “ac” is here to be replaced by the achievement’s name.

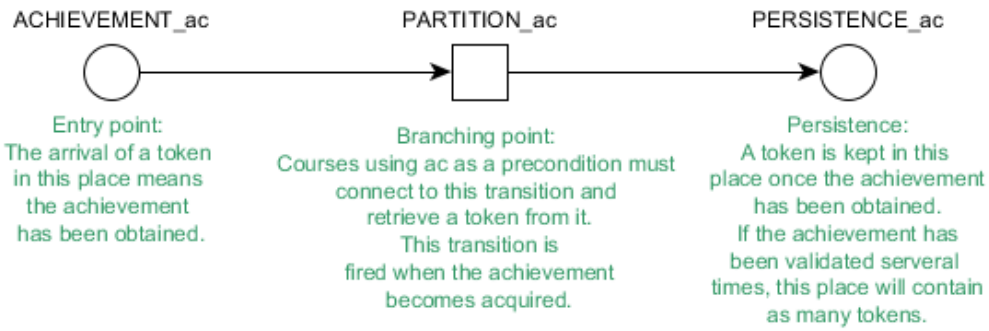

Figure 5: Petri Net modelling of an achievement

e. Courses

Figure 6 shows how an example course is translated into Petri Net. This example presents a course whose identifier is “c”, with achievements “pre1” and “pre2” as preconditions, and granting achievements “ac1”, “ac2” and “ac3”.

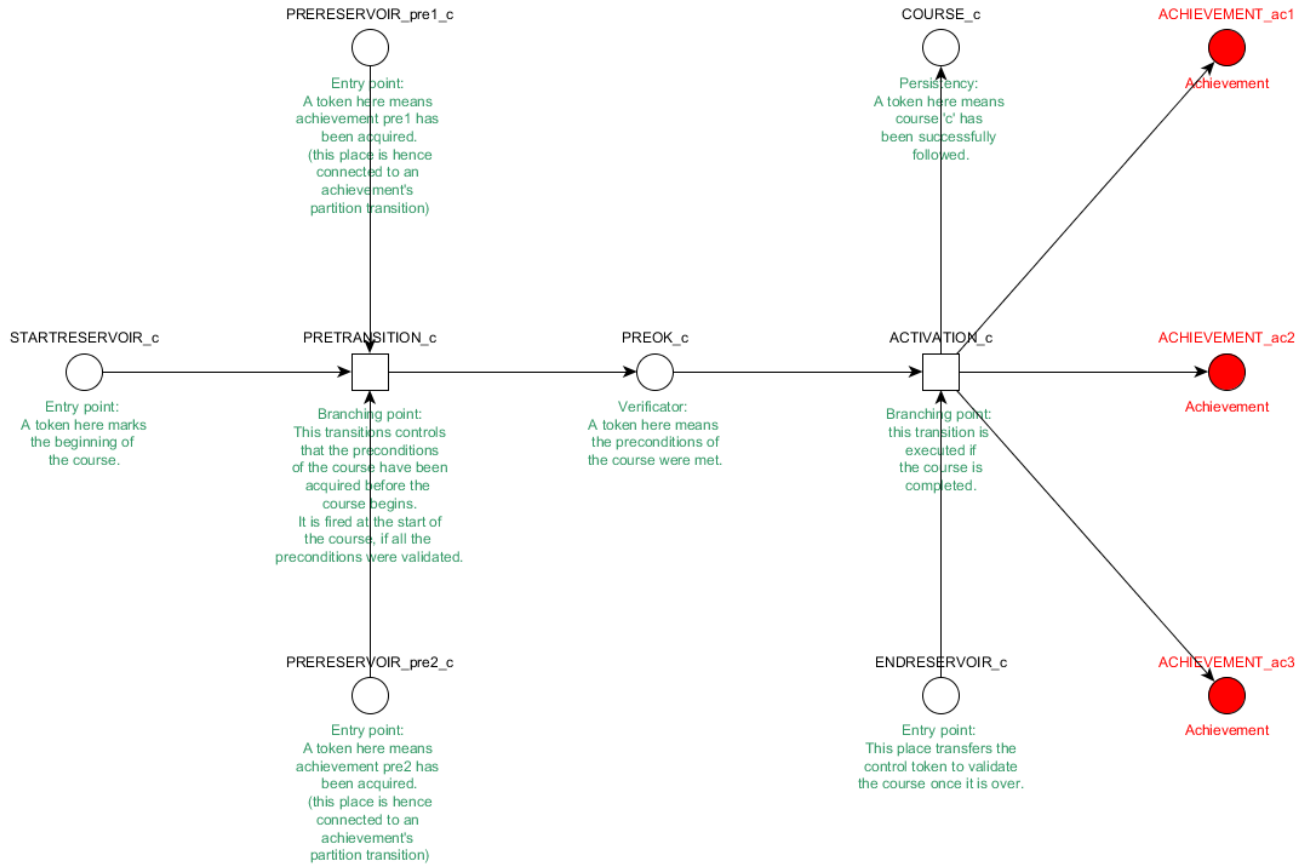

Figure 6: Petri Net modelling of a course (example)

As for the way courses are integrated into the Petri Net and interact with other entities, the control token is used twice here.

The first time a course receives the Control Token is when its preconditions are checked. The control token then arrives in the STARTRESERVOIR_ place, and is consumed by the PRETRANSITION_ transition if the preconditions of the course are met (that is, if each of its PRERESERVOIR_ places contains a token). The PRETRANSITION_ then retransmits the Control Token to the next entry point place needing it. This next place can be the entry point of the tick following the start tick of this course, the STARTRESERVOIR_ place of another course beginning at the same tick as this one, or the ENDRESERVOIR_ place of another course ending during this course's start tick.

The second time the Control Token gets into a course's construct is when it arrives to the end tick of that course (we already mentioned this fact in the previous paragraph). The Control Token in then transferred into the ENDRESERVOIR_ place of the course. The course's ACTIVATION_ transition then sets all its achievements as acquired if the PREOK_ place contains a token, that is, if the preconditions of the course were met during its beginning tick. The PREOK_ transition then transmits the Control Token in pretty much the same way than the PRETRANSITION_ did. The only difference is here that the Control Token can't be transmitted by ACTIVATION_ to a STARTRESERVOIR_ place. Hence, it is necessary that the verification first transmits the token to all the courses beginning in the current tick, and only then, sends it through each of the ending courses. If this were not so, it would be possible for a course to validate a precondition obtained from another course ending at the same tick this course begins at, depending on the order in which the courses are treated. As we have already said, we made the assumption in our model that this must not be possible. Hence the fact we impose this order of treatment.

f. Co-courses

Figure 7 shows how a co-course link is translated into a part of the Petri Net. It presents the example of a course named “c”, taking course “co1” and “co2” as co-courses.

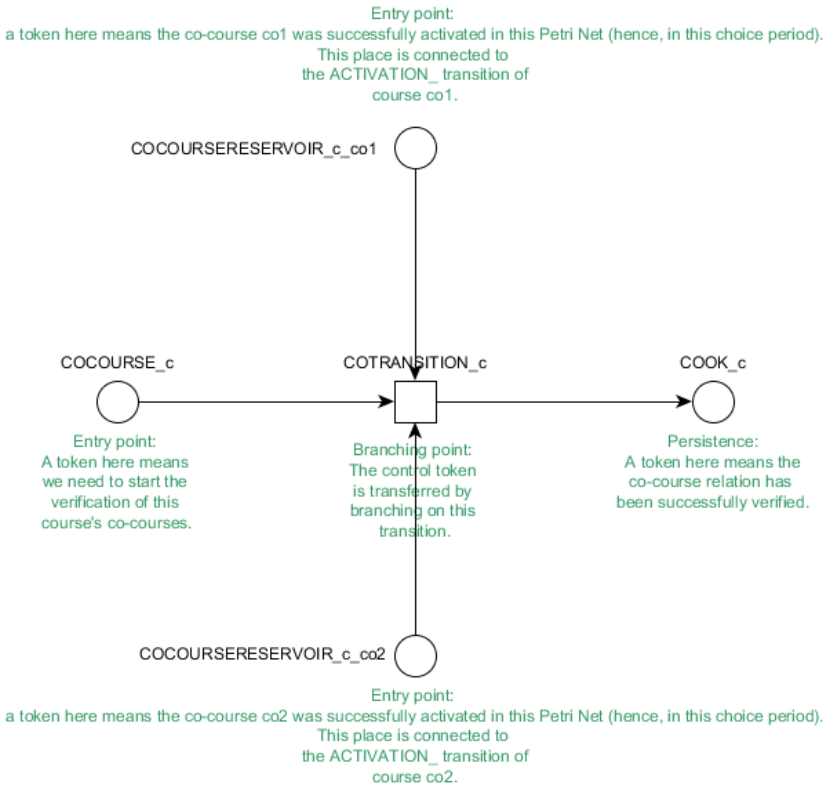

Figure 7: Petri Net modelling of co-course constraints (example)

As with courses, the verification of co-courses is achieved by making the Control Token “flow” through each of the co-courses. However, in the case of co-courses, each co-course is crossed only once by the Control Token. As shown on the figure, each course checks all of its co-courses at once. The Control Token arrives in the COCOURSE_ place of the course. The COTRANSITION_ transition is then activated if all of the course’s co-courses have been validated during this choice period, which is true if all the COCOURSERESERVOIR_ contain a token (this is achieved connecting the COCOURSESRESERVOIR_ places to the ACTIVATION_ transition of the corresponding courses). Once the execution of the Petri Net passed the verification, the Control Token is transmitted to the next co-course verification’s COCOURSE_ place, to allow its verification, or, if no other co-course is to be checked, to the COCOURSESOK_ place (see figures 1 and 2 for an example) by the COTRANSITION_ transition.

g. Program

The way a program is modelled in Petri Net is presented in figure 8. This figure shows the example of a program named p, needing achievement ac1, ac2 and ac3 to be validated.

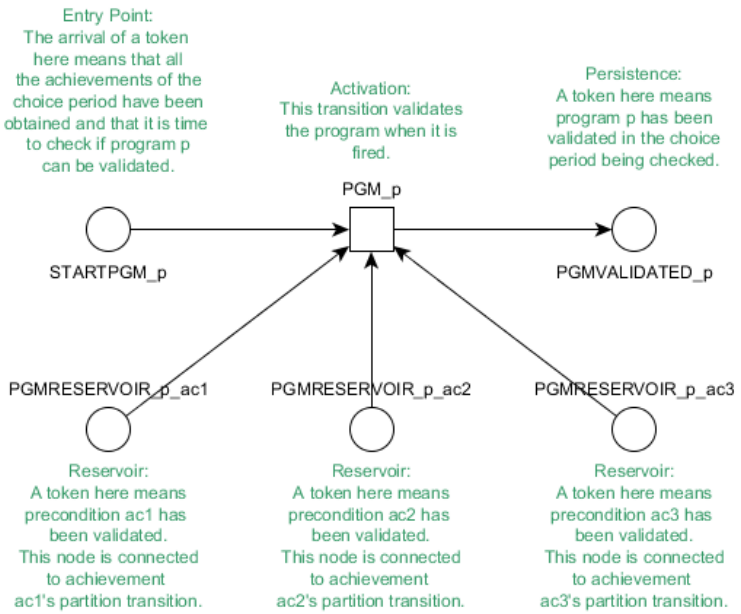

Figure 8: Petri Net modelling of a program (example)

Regarding the way program completion is integrated to the global Petri Net, it is important to notice that, unlike the verification steps, it does use the blocking of the control token to detect abnormalities. Indeed, no abnormality is possible here, as we only want to see whether some available program is completed with the newly obtained achievements. This implies that the Control Token is not transmitted to the programs, but rather triggers, via the ENDTRANSITION_ (see figure 1 and 2) the sending of a token to each of the programs so that they can all be checked together and do not block the verification.

5. Possible uses of the model

We explain here how the model can be applied to a real case. We detail first the use of the model in its full possibilities. Then, we give a more realistic view of it, taking the limitations and constraints of "real" deployments into account.

a. Theoretical use

We already gave hints about the full use of the model in the preceding sections.

The idea in this use of the model is to apply it strictly: defining in advance for each course a detailed list of all the achievements it offers. Hence, it is also necessary to define for each of the programs a complete list of the achievements (that is, the skills) it needs to be validated. It is also necessary to precisely define the preconditions of each course, as sets of achievements granted by other courses.

As one can easily figure out, standardizing achievements is not an easy task. Indeed, this requires that all the courses either needing or granting a specific piece of knowledge refer to it by the means of the same achievement. This can be extremely tricky in big organizations, with a long-defined system needing to be converted into the model.

b. Realistic use

As we have seen, the theoretical model encounters some limitations when it comes to adapting already established systems.

Such systems, however, already use the notions we defined, but they do it in a less detailed manner. Indeed, preconditions of a course are then defined as the courses that a student needs to have followed to validate that course. Same goes for programs: a program is then no more completed with a set of achievements, but a set of courses.

As we already made clear in the previous paragraph, this situation replaces the concept of achievement by the course one. An easy way to realize this with our model is to simply associate each course with a unique achievement.

Of course, it is also possible in this system to have a bit more complicated situations. Indeed, in some cases, courses may specify a kind of alternative precondition: saying that only one course of a given set needs to have been followed. In the original model, this would mean that each of the courses of the set grants, possibly among others, a given achievement. In the case were achievements are uniquely associated with courses, this cannot be done as-is. The solution to that is to kind of mix this situation with the theoretical use, by adding several achievements. Indeed, adding a common achievement for all the courses needed in such a set of precondition courses (and thus making this achievement model the set of courses) as a precondition for the course solves the problem. This solution keeps the model simple and allows full compatibility with existing systems.

As this realistic model is nothing but a simplification of the theoretical one, everything applying to the latter also applies to the former.

6. Illustration

This section gives very simple examples of different key situations to understand the way the Petri Net modelling works.

Remember the numbers in the places (the circles) indicate the number of tokens contained in these.

a. Typical example

Let us begin going back to the example shown in figure 1. Figure 9 shows the state of the Petri Net presented in figure 1 after full execution of the Petri Net. The colors on the figure have also been kept from figure 1, for more clarity.

Of course, this execution result considers that, before starting the propagation of the tokens, the Control Token has been inserted in the place pointed out as its starting point in figure 1. Hence, STUDYPERIOD_period1 must be populated with one token before execution of the net begins in order to get this result.

Let us now briefly analyze the result of the propagation, and the meaning of the tokens located in it.

As we can see, the END_ place contains a token. This means the Control Token has encountered no abnormality while flowing through the net, and hence that the proposed study program is correct from a preconditions and co-courses point of view. As the program does not contain any abnormality, it means that all the ticks were successfully traversed, and so explains that each TICKDONE_ place contains a token, maintained there to express this information. In the same idea, each course contained in the study program has thus been validated, implying that each course's COURSE_ place contains one token, again, for persistence of information. Same goes for COOK_ places, which make persist the fact that co-courses were successfully verified for the course they are related to. Each of the validated achievements also keeps this information by stocking a token into its PERSISTENCE_ place. These PERSISTENCE_ places contain as many tokens as the number of times the achievement has been validated. In our example, achievement ac1 is validated by both course1 and course2, and hence PERSISTENCE_ac1 contains 2 tokens.

The last information we can get from the tokens concerns the completed programs. As we can see here, two programs are available to be completed: p1 and p2. However, p2 needs achievement ac4 to be acquired. This achievement is not acquired in the study program, hence contains not token and blocks the token arriving to p2 in the STARTPGM_p2 place. On the other hand, p1 has been completed, and node PGMVALIDATED_p1 thus contains a token.

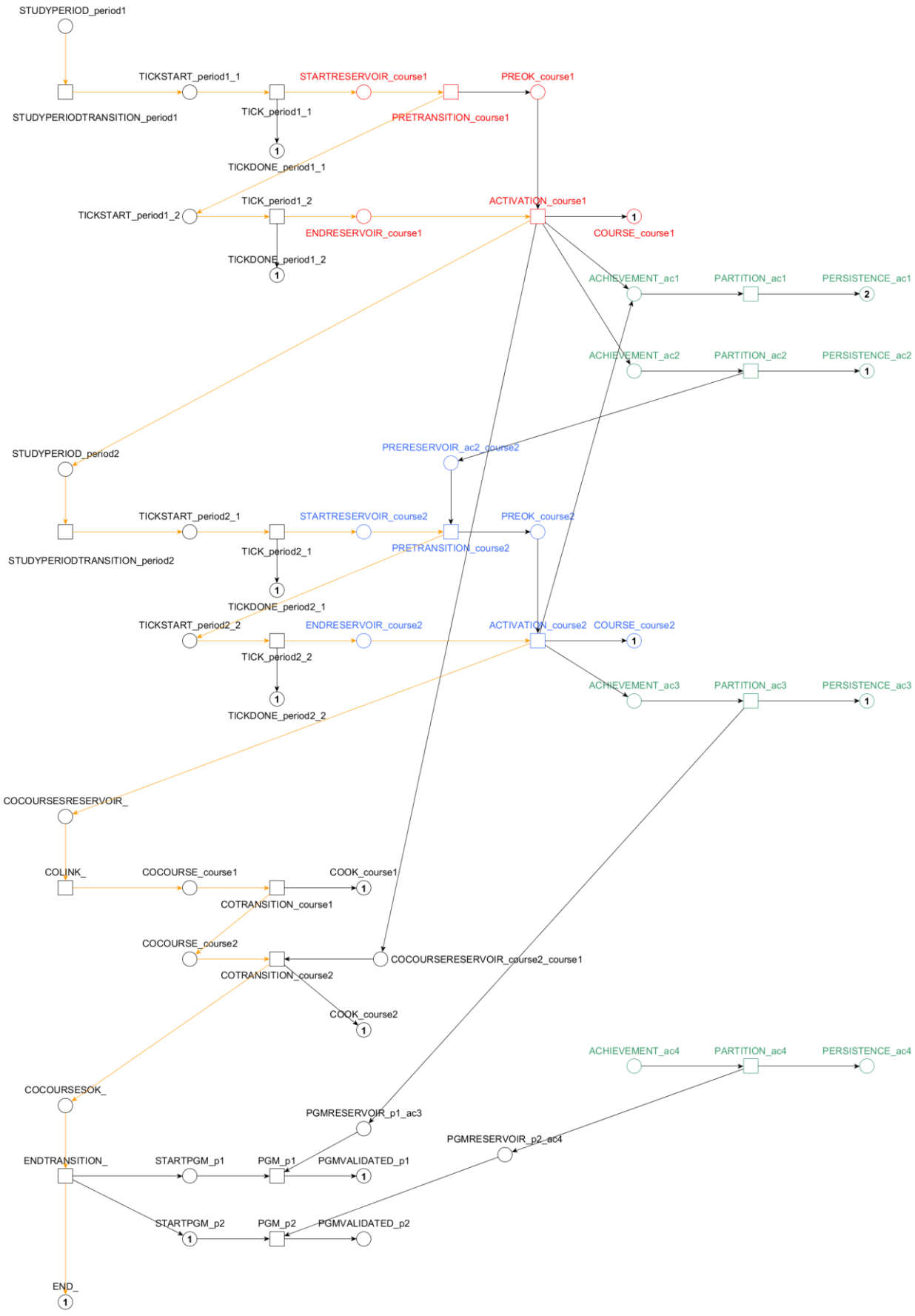

Figure 9: Figure 1's Petri Net after full execution (see appendices' archive for full size)

b. No precondition

Of course, it is possible for a course not to require any precondition to be fulfilled. The Petri Net modelling allows this in a very simple way: if a course does not have any precondition, we just do not need to link anything as entering into the PRETRANSITION_ of this course.

An example of this is given in the example of figures 1 and 9, with course course1 (in red on the figure). As we can see there, PRETRANSITION_course1 is not linked to any achievement, and hence does not block the verification (as the Control Token does not have any constraint to cross it). This models the fact course1 has no precondition.

c. No co-course

A course is also allowed not imposing any co-course constraint. This is achieved in the same way as for preconditions, as explained in the previous point.

The example of figures 1 and 9 show both the case of a course with one co-course, and one with no co-course. Indeed, course2 has course1 as its co-course, as course1 does not have any co-course.

As we can see with course 1, the situation in which a course does not have any co-course is modelled not linking anything to this course's COTRANSITION_. In our example, COTRANSITION_course1 is hence non-blocking for the control token.

d. Missing preconditions

Let us now illustrate how a non-validated precondition impacts the Petri Net execution and how this deficiency can be tracked after executing the net.

Figure 10 shows the initial modelling of a course named course 1 taking achievement ac1 and ac2 as its preconditions and granting achievement ac3. Only the modelling of the course is shown here, with orange labels to underline the path of the Control Token, as the rest of the net is not concerned by what we intend to show here.

As we can see on this figure, ac2 has already been acquired, but this is not the case for ac1. The verification of this course should hence trigger an error message informing the student trying to submit this study program of this abnormality.

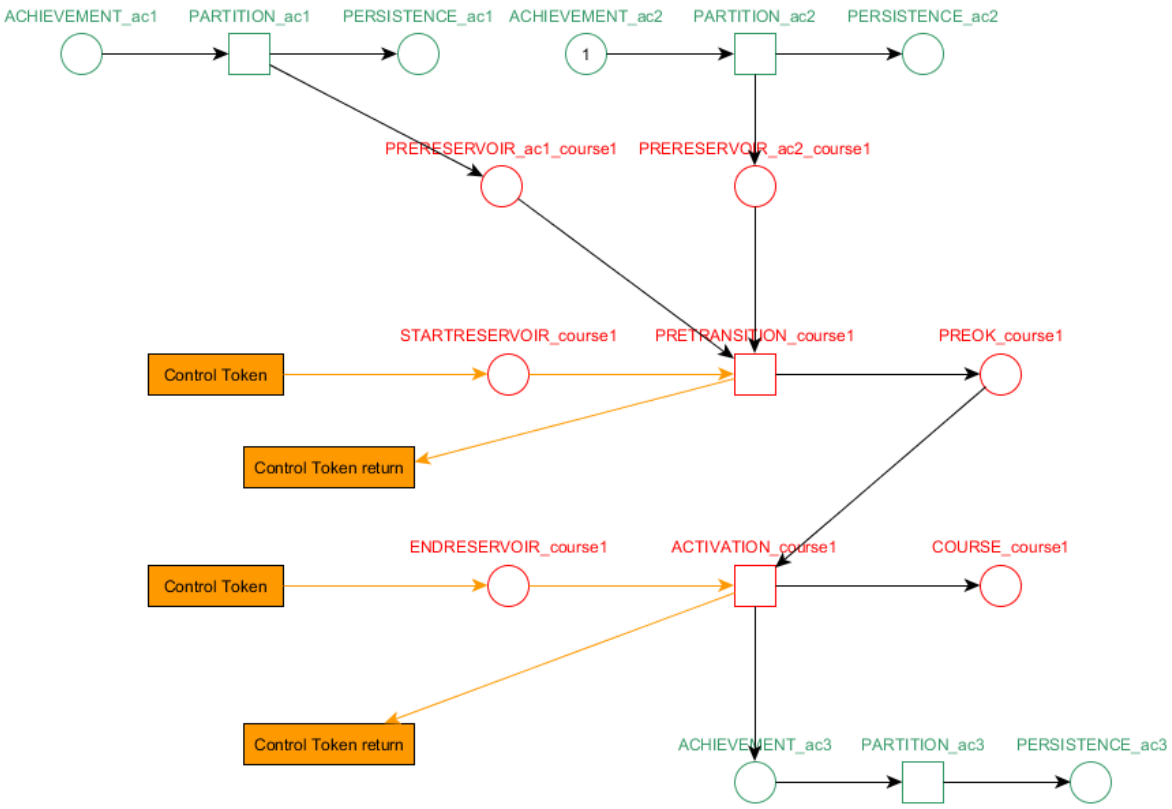

Figure 10: Course example with a non-validated precondition, before execution

Figure 11 shows the result of the execution of the Petri Net part presented in figure 10.

This figure contains an arrow in bold, showing in which part of the Petri Net the Control Token gets blocked because of the non-validated precondition.

Indeed, as we already mentioned earlier in this document, an abnormality in the preconditions of a course causes the Control Token to stop its journey through the net. The verification algorithm then can post-process the fully executed Petri Net, and see where the Control Token is located to infer what the problem is. As we can see in our example, the Control Token is stuck in the STARTRESERVOIR_course1 place, because the PRETRANSITION_course1 transition cannot be fired. This transition, in turn, cannot be fired because PRERESERVOIR_ac1_course1 does not contain any token, due to the fact achievement ac1 was not acquired, i.e. ACHIEVEMENT_ac1 did not receive any token, and transition PARTITION_ac1 was not executed.

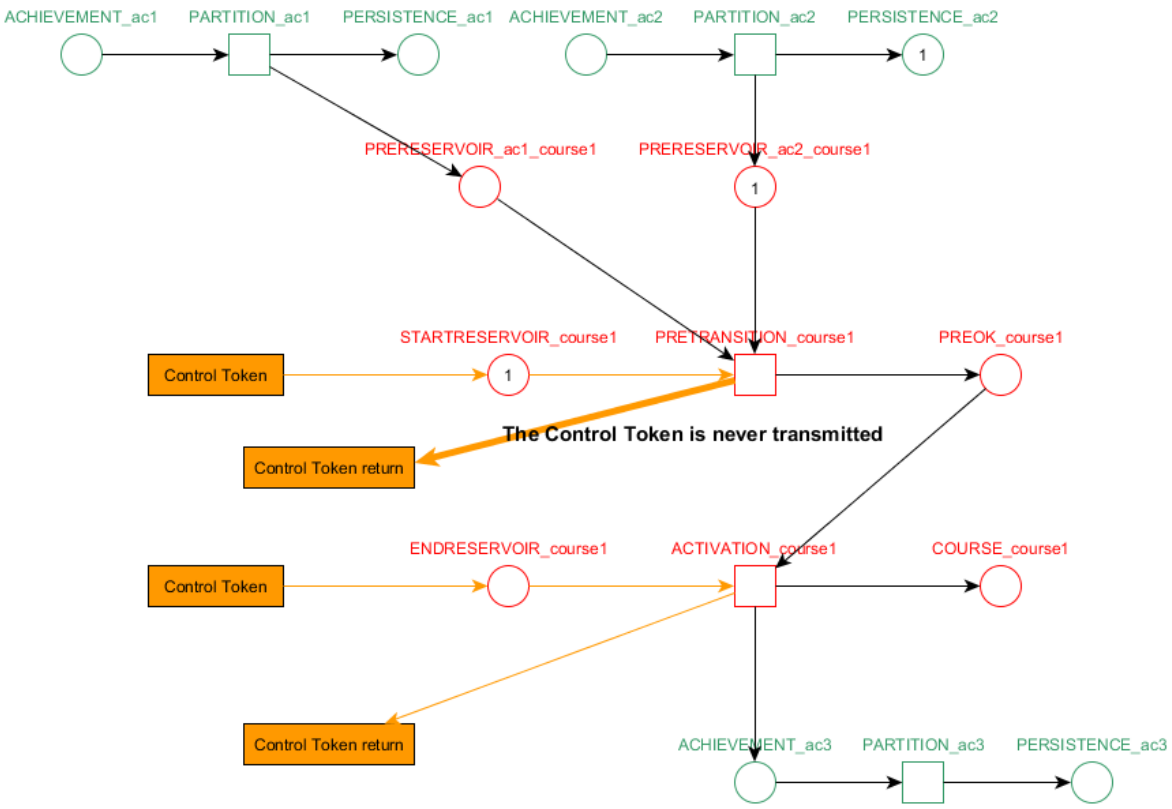

Figure 11: Figure 10 after execution

e. Error in co-courses

As in the previous point with preconditions, we now describe how missing co-courses are managed by the Petri Net modelling.

Figure 12 shows the example of a course named course1 taking both courses course2 and course3 as its co-courses. In this case, course3 has been chosen in the same choice period, but not course2. As in the previous figures, a red rectangle corresponds to a link to another part of the Petri Net, not represented here. Orange arrows and labels mark the path followed by the Control Token.

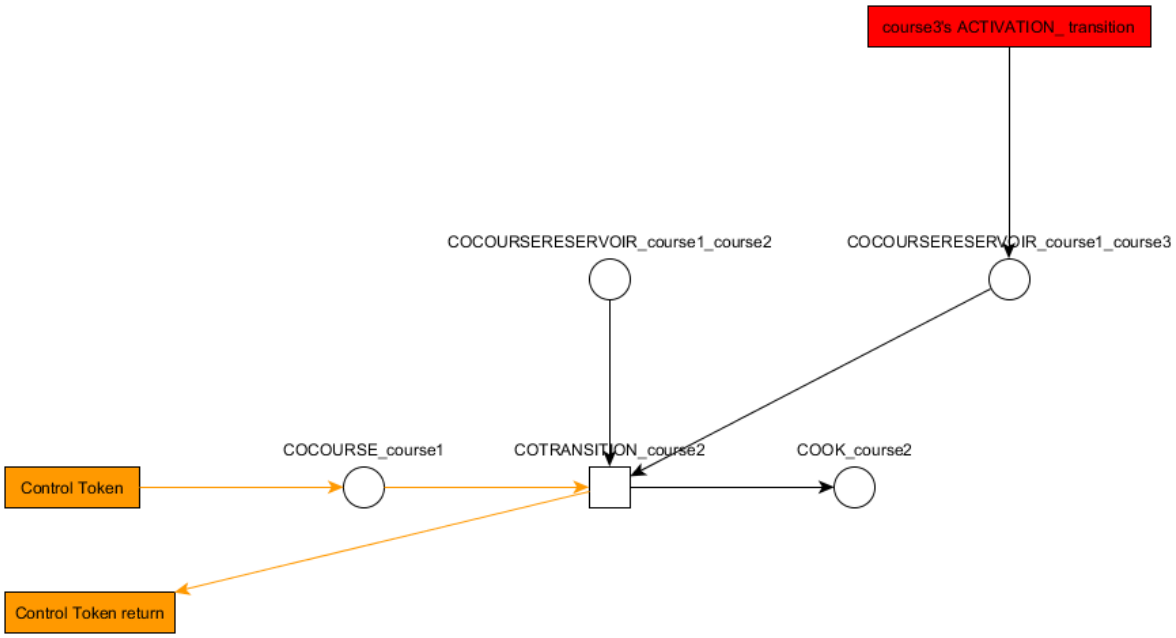

Figure 12: Course example with an error in co-courses, before execution

Figure 13 shows the result of executing figure 12's Petri net.

As we can see, just like for preconditions verification, the Control Token is stuck. Indeed, as course course2 was not selected, no token arrives to place COCOURSERESERVOIR_course1_course2, causing the COTRANSITION_course2 transition to never get fired, and forbidding it to transmit the Control Token any further.

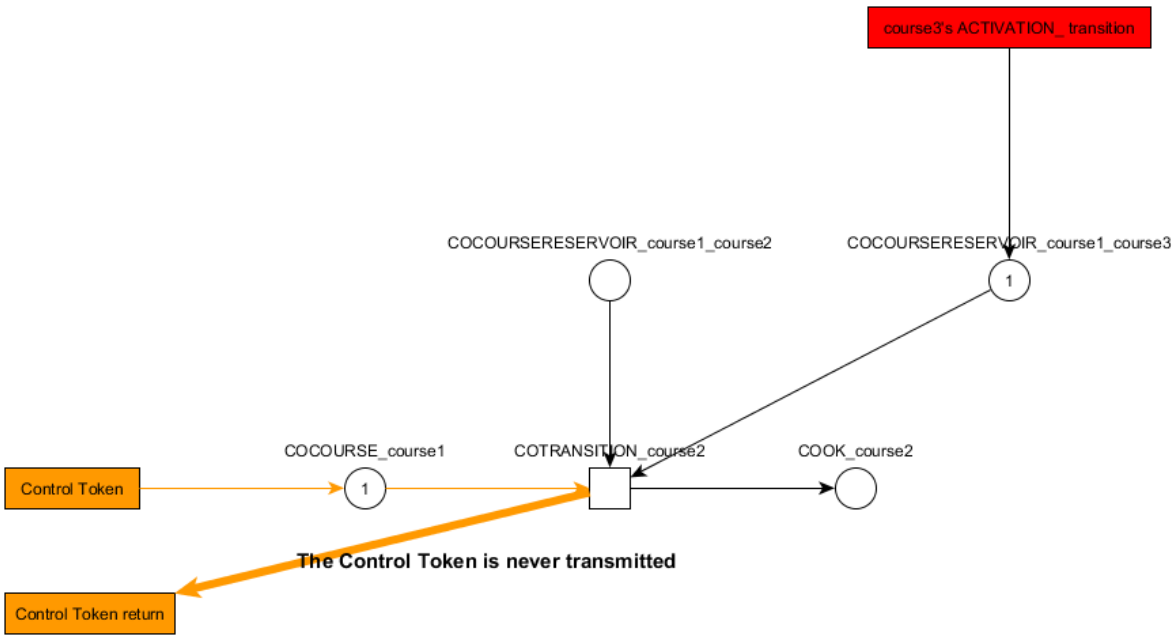

Figure 13: Figure 12 after execution

f. Taking the previous choice periods into account

With the modelling we explained until now, it is only possible for a course's preconditions to be validated if the achievements corresponding to them have been acquired thanks to some other course. This is a major limitation, as our modelling considers only one choice period at a time, causing the fact that only the achievements granted by courses selected in the choice period being verified will be available to check preconditions. Hence, as it is until now, the presented modelling ignores the courses selected in the previous choice periods.

Of course, this limitation is unacceptable. But fortunately, there exists a simple solution, reusing part of the modelling we already have. It simply consists of adding each of the achievements acquired thanks to the courses validated in the previous choice periods and inserting a token in their ACHIEVEMENT_ place so that they behave exactly as is they had been acquired during the current choice period.

An example of this principle is shown in figure 14. It presents a course in the Petri Net needing two achievements (ac1 and ac2) as its preconditions. Let us consider these achievements are not granted by any course in the current choice period but have both been acquired in the past. They are hence injected into the net as we explained.

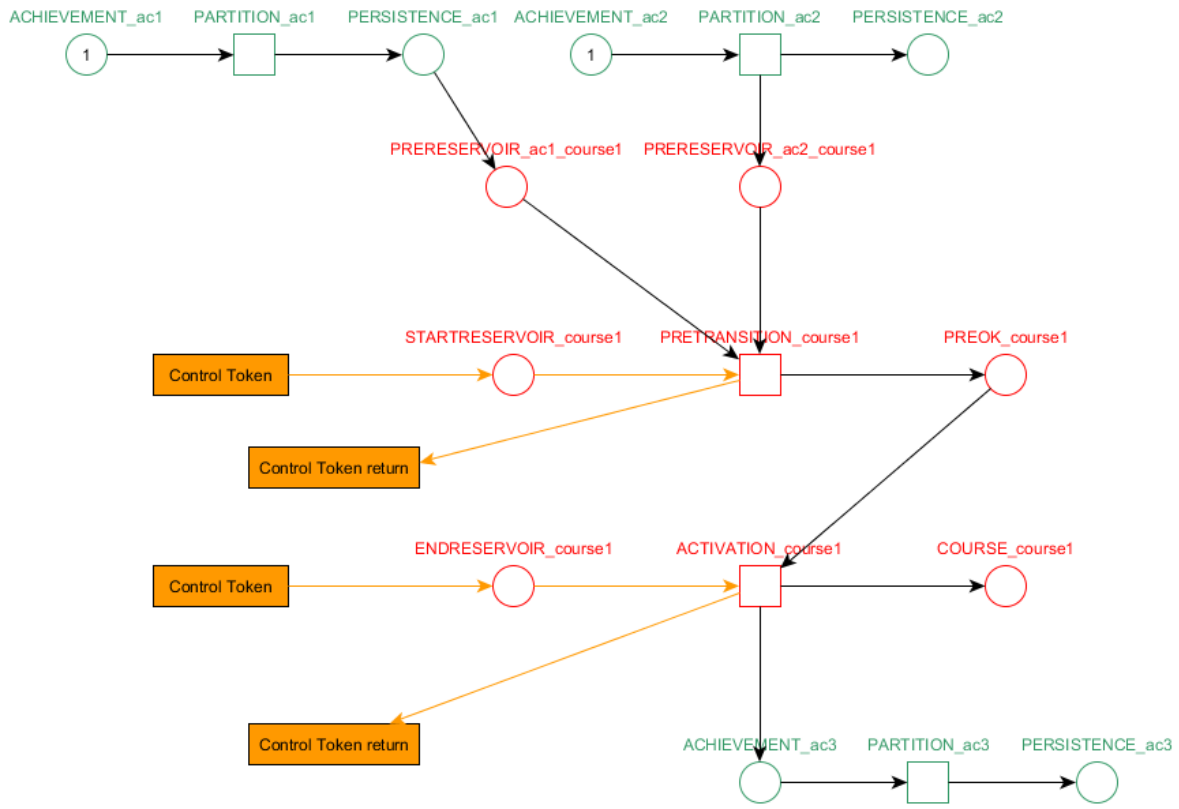

Figure 14: inserting previous choice periods' achievements

7. How verification is achieved

Let us now take a glimpse at the way our Petri Net model is used to verify study programs.

a. Conversion from original data to Petri Net

The complete conversion of a study program into a Petri Net is achieved by the *coursescoord.verification.PetriCourseBuilder* class, which contains all the static methods implementing the conversion algorithm.

A complete pseudocode version of the conversion algorithm can be found in the documentation (both in the code and generated Javadoc) of method *buildPetriNetFor*, in that class.

b. Verification algorithm

The verification algorithm is fully implemented through static methods in the *coursescoord.verification.PetriCoursesAnalyser* class. The documentation (again, both in the code and Javadoc) of the *analyseProgram* method of this class contains a pseudocode version of the algorithm.

Note that we will refer to abnormalities in the study program as errors when referring to the verificator. Indeed, the verificator only considers as "correct" a study program containing no abnormalities at all.

The general idea of the verification is to first trace precondition errors in the Petri Net (that is, cases in which some place linked to the PRETRANSITION of the course did not receive any token and hence forbid the transition to be activated), and then check the co-courses constraints.

Preconditions verification is actually achieved quite simply. The program just tests that each TICKDONE place contains a token, by going through these nodes by chronological order of the tick they belong to. Once it finds a place that does not meet this condition, this means that its TICK place did not receive the control token; hence, the control token got blocked in some PRETRANSITION during the previous tick. The algorithm then checks the previous tick, by going in the order they were placed after the TICK place through each of the PRETRANSITION transitions. Each time it finds a PRETRANSITION that did not execute and blocked the token, it reports a precondition error telling which achievements blocked the transition (this information can easily be obtained by checking the PRERESERVOIR places of that transition), and then adds the missing token before executing the Petri Net again, so that the verification continues as if no error had occurred and the algorithm is able to find possible errors in what follows in the net. This is the same idea than the one applied by programming language compilers: when you find an error, report it, and then correct it so that you can find additional errors in the rest of the code (even if, unlike here, in the case of compilers, correcting the error is not always possible).

Then comes the co-courses verification. It basically works in the same way than the preconditions one. The algorithm checks all co-courses one after the other by checking that the COOK place contains one token. If it is not the case, a co-course error is reported and the COTRANSITION's COCOURSESRESERVOIR places are checked in order to see which co-course constraints were not satisfied. As for preconditions, once an error has been found, it is corrected before proceeding to the rest of the verification.

After these two steps, warnings are generated, as we will discuss later.

The last point of the verification consists in checking which programs were completed by the study program. For this, the algorithm simply checks the PGMVALIDATED nodes, but this is not all. Indeed, the programs contained in the Petri Net were only added if they were available for the student, that is, if the student had completed the required programs for them to be available. Hence, if a program is completed by a student, it is possible that it unlocks some other program that could *immediately* be completed too, because the student already has acquired all the achievements that program needs. This is why, when a program is discovered as completed, the algorithm checks which new programs are available after that, which were not before. If it finds some, it adds them to the verification Petri Net and reinjects a new control token in the program validation region of the Petri Net, as well as in all the ACHIEVEMENT places of the achievements the user has already acquired (that is, the ones having a PERSISTENCE place containing one or more token). The Petri Net is then re-executed, and the program verification part of the verification algorithm recursively called on it, until no new program is completed by the student. It is very important that the program verification be the last step of the verification, as it can heavily modify the Petri Nets (due to the way it injects tokens), and cause the following steps to get flawed results.

c. Error notifications

Error notifications are generated by preconditions and co-courses errors. In the program, they are modeled by the *coursescoord.verification.ErrorNotification* class.

An error notification corresponds to something blocking the verification in the Petri Net if not patched. They correspond to abnormalities in the study program that the student should either justify or correct, as we explained in point 2 of section 3.

d. Warnings

Warnings are a less critical level of error messages than error notifications, also generated when checking a study program. They are not shown to the student, and only the commission members are able to see them when receiving a study program proposal. As their name suggests, they basically are messages reporting something that seems a bit weird and might cause the commission to refuse a proposal.

In the current version of the application and verification algorithm, only one type of warning can be generated: if an achievement has been validated a greater number of times than a given threshold, which might indicate that the student is selecting too many courses granting the same skills, and hence does not learn enough for the program to be accepted. These warnings

are simply generated by iterating on the PERSISTENCE places and checking that their token count is under the value defined as the MULTIPLE_ACHIEVEMENTS_THRESHOLD constant in *coursescoord.verification.PetriCoursesAnalyser* (this value is set to 2 in the provided code, for test purposes. A greater value would be more interesting in a real system deployments; this has been done so here in order to make warnings possible with the test data). Of course, future versions of the application could perform this verification on a more "dynamic" value.

e. Error suppression

Some of these error notifications may have been marked by the commission to be ignored at verification for a given study program (for example, if a student is editing a study program for which there were several abnormalities, but the commission accepted only part of them). This is why, after the verification itself, the set of obtained error notifications is checked so that we remove all the errors corresponding to already accepted abnormalities from it.

6) Going further

Now that we have presented the application in depth, let us recapitulate the points that make it easier to use and extend.

1. Portability of the application

As we have told, the application has been developed in Java EE 7 in order to run on a Glassfish 4 server. As Glassfish strictly implements the Java EE specification, this means any other server implementing the JEE 7 specification can run the application, whatever additions it makes to it. Hence, if a Java EE server is already deployed in the system the application must be used with, it can directly integrate it without needing the computer scientists responsible for it to deploy another server.

2. Portability of the modelling

We have seen that the model had been developed to be as easy to integrate and customize as possible.

An example of this portability resides in the way data access is managed, with the drivers system, which allows complete flexibility for it. As an example, we mentioned several times the provided test database, but nothing forbids the developer wanting to integrate the application to his system to implement drivers in such a way that they only access files instead of a database, or even to hardcode some data into them, if he considers it more appropriate for his use of the system.

Another example consists of the way Ticket and TicketPart objects interact. As we have seen, it is very easy to add new kinds of ticket parts into the system, as it has been design to allow these modifications to be dynamically taken into account in the pages needing to display them.

3. Portability of the Petri Nets verification

The Petri Net model used by the application to verify study programs is also one of its biggest positive aspects in terms of ease of extension.

Indeed, as we have shown, these Petri Nets are divided in several “conceptual” zones, corresponding to different parts of the verification, each of them triggered in turn by the passage of the control token. It is thus very easy to add some new zone in case some additional verification should be performed at the Petri Net level.

The BlindPN library also offers the possibility to export or import Petri Nets using XML files. This is not used in the application (yet it proved very useful for debugging), but could be an interesting tool to make the nets persists if need arises to when extending the system.

7) Possible improvements

As it is always the case, the application is improvable. We list here some interesting ameliorations that could be made on it.

1. Linguistic support

The system has entirely been designed in English. Although this is only a detail, a more generic linguistic support would of course be a good point. This could be achieved using an application bean (that is, roughly, a globally-available object) responsible to access all the linguistic resources. Whenever the application would need to issue some message, it would then call this bean and ask it for the message in the right language, based on the language choice made by the user in some menu.

2. Dynamic time division

As it is now, the choice periods are assumed to contain always the same study periods in the same order, and these periods to contain all the exact same amount of ticks. This works for a vast majority of uses of the system, but maybe some cases could be found in which a less "static" approach could be better.

Hence, a possibly interesting extension of the system and model would be to allow the choice and study periods' content to vary.

3. TextOnlyTicketPart

We have mentioned that the system contained an unused, yet functional, implementation of the TicketPart class: TextOnlyTicketPart. Of course, it could be interesting to integrate it in a better way into the system.

4. End user's comment

This section's previous points gave comments from a developer's point of view about additions that could be made to the application. Of course, these comments are not really the most important, as the crucial point of such application is the way users will actually be able to use it. This point hence presents the general comments given by Ms. Catherine Dumont, UCL's philosophy, arts and letters faculty's assistant director. It thus gives a "commission-oriented" point of view on the application.

Ms. Dumont proved globally pleased by the application. She noticed the following points, from the most important to the less significant one, as useful improvements to implement into the system:

- As it is, the application does not take credits (ECTS) associated to courses into account. They should be added in order to group all the verification's parameters on the same platform. Also, verifying the number of acquired credits should become part of the verification.

- A “teacher” access to the platform could be useful in order for the commission to directly communicate with teachers of the concerned courses when receiving a study program proposal presenting some abnormalities. For example, a student could want to follow an Ancient Greek course, but propose a study program asking to directly follow the second year’s course, as he already had a strong formation in Ancient Greek in secondary school. The commission might then find it useful to ask to that course’s teacher whether he thinks it is a good idea or not before taking the final decision and communicating it to the student. Allowing it directly in the platform could thus be very useful.
- Adding a courses suggestion feature with failed courses (hence, courses of the previous year that a student must follow again this year because he failed at the exam) would be a good idea, as these courses are most likely to be needed by lots of other courses.

8) Conclusion

This document presented a study program verificator application based on Petri Nets, developed as a master thesis in SINF22 at UCL.

We detailed here the different functionalities of the application, as well as its programmatic design and the choices made in order to implement all that. Apart from that, we also described the Petri Net model that was created to perform the verification, and the way we use it in order to generate either error or warning notifications to the user in case something seems wrong with a study program. Finally, we gave some amelioration possibilities for the application.

Appendices to this document are the following:

- An additional document containing
    - A complete installation guide to compile the application and make it run locally for testing.
    - A user guide explaining how to use the application.
    - A description of the data contained in the test database.
- An archive file containing
    - A dump script of the test database to regenerate if. (**testData.backup** file, at the root of the archive)
    - The complete source code of BlindPN (**blindpn** directory)
    - The complete source code of the application (**coursesCoordinator** directory)
    - Precompiled versions of the application and BlindPN, to ease deployment (**precompiled** directory)
    - A developer guide, in the form of Javadoc to generate from the source codes (see installation guide for instruction)
    - All the figures used in this documents, in full resolution, so that they are more readable (**figures** directory)
    - A README file containing a description of the archive's content.

9) Bibliography

About Java EE:

[1] http://docs.oracle.com/javaee/7/api/

[2] http://openclassrooms.com/courses/creez-votre-application-web-avec-java-ee

[3] http://java.sys-con.com/node/1984284

[4] http://www.mkyong.com/jsf2/access-a-managed-bean-from-event-listener-jsf/

[5] http://incepttechnologies.blogspot.be/p/view-parameters-in-jsf-20.html

[6] http://stackoverflow.com/questions/25694423/is-there-any-way-to-pass-an-object-between-viewscoped-backingbeans-without-using

About Java 8:

[7] https://docs.oracle.com/javase/8/docs/api/

[8] http://www.techempower.com/blog/2013/03/26/everything-about-java-8/

About JPA and Eclipselink:

[9] http://www.objectdb.com/java/jpa/query/jpql/collection

[10] http://stackoverflow.com/questions/2772305/jpql-in-clause-java-arrays-or-lists-sets

[11] http://www.kianworknotes.com/2012/12/jpql-in-clause.html

[12] http://stackoverflow.com/questions/2488930/passing-empty-list-as-parameter-to-jpa-query-throws-error

[13] http://blog.denevell.org/java-jpa-postgresql-jersey.html

[14] https://glassfish.java.net/javaee5/persistence/persistence-example.html

[15] http://stackoverflow.com/questions/17078412/jpa-persistence-xml-meta-inf-not-working-correctly

[16] http://en.wikibooks.org/wiki/Java_Persistence

[17] http://www.tutorialspoint.com/jpa/jpa_orm_components.htm

[18] http://www.vogella.com/tutorials/JavaPersistenceAPI/article.html

[19] http://www.objectdb.com/java/jpa/entity/persistence-unit

[20] http://javahowto.blogspot.be/2008/10/helloworld-with-eclipselink-and-mysql.html

[21] http://www.eclipse.org/eclipselink/documentation/2.6/jpa/extensions/toc.htm

About PrimeFaces:

[22] http://www.primefaces.org/showcase/

[23] http://www.primefaces.org/docs/guide/primefaces_user_guide_5_1.pdf

About Glassfish:

[24] https://glassfish.java.net/docs/4.0/installation-guide.pdf

About Maven:

[25] http://maven.apache.org/download.cgi

[26] http://maven.apache.org/plugins-archives/maven-archetype-plugin-1.0-alpha-7/examples/webapp.html

About Petri Nets frameworks:

[27] https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html

About OTRS:

[28] https://www.otrs.com/otrs-business-solution-for-better-customer-service/otrs-as-customer-service-software/

About PostgreSQL:

[29] http://www.postgresql.org/

Additional tools used:

[30] http://www.yworks.com/en/products/yfiles/yed/

[31] http://projects.laas.fr/tina//

[32] http://www.jedit.org/

10) Acknowledgements

M. Marc Lobelle, my thesis advisor, for his guidance through the whole conception of the application.

Ms. Catherine Dumont, UCL's philosophy, arts and letters faculty's assistant director, for her help defining the requirements of the application and testing it.

M. Edmond Mulemangabo, researcher at UCL, for his help entering the Java EE universe, and being one of this thesis' official readers.

Ms. Chantal Poncin, study advisor, for her help gathering the requirements to fulfill for the application, and being one of this thesis' official readers.

M. Charles Pecheur, teacher and researcher at UCL, for being one of this thesis' official readers, and his "Concurrent Systems" course (LINGI2143), without which I would not have known anything about Petri Nets.

M. Baudouin Le Charlier, for his LSINF1160 and LSINF1161 courses, which learned me to program, made me discover Java, and more importantly gave me my development philosophy.

UCL in general, for the last five years I spent in it.

The SINF21, SINF22, INFO 21 and INFO22 students in a whole, for their help, support, and silly jokes.