System Modelling and Design
From B to OO Classes:
Transforming B Machines to Classes

Revision: 1.5, August 29, 2005

Ken Robinson

2nd September 2005
mailto::k.robinson@unsw.edu.au

©Ken Robinson 2005

Contents

1 Modelling Classes 2
   1.1 Important parts of model to note 3

2 From B to UML: A Case Study 4
   2.1 A Program Manager 4
   2.2 A Note on Modelling 4
   2.3 A Class Diagram 5
   2.4 Notes on B Objects 5

3 Implementation 6
   3.1 Augmentation of machines 6
   3.2 Implementing Class Attributes 6
   3.3 Notes on the form of guards 6
   3.4 Proof Obligations 6
   3.5 Implementation of ProgramManager 7
   3.6 ProgramManagerDB base 7
   3.7 Incremental implementation 8

4 Invariant in an Implementation 8

5 Invariant and Variant in a while construct 9

6 Using Strings 9
Objectives of this Lecture

This presentation has two objectives:

1. outlining a path from a the B Method (B) development (a set of machines) to OO classes. At the first level, the classes are abstract in the sense that types may be relations, and hence not types that would be supported by a typical OO programming language.

2. illustrating an implementation strategy for such classes.

1 Modelling Classes

B has no intrinsic support for classes. B machines may be thought of as objects, that is as instantiations of a class. B is sometimes referred to as “object based” rather than “object oriented”.

We will start by considering the exercise of modelling a class using B.
Consider the class $\text{OO}$, which we simplify by showing only 1 attribute and 1 method.

```java
public class OO {
    public OO(cons_args);
    public attr_i_type attr_i;
    public method_i_type method_i(args) {
        body_i;
        return result_1;
    }
}
```

Remember that “class” is a synonym for “set”: a class represents a set of objects. Thus we model a class by a machine that manages a set of objects:

```
MACHINE OOClass
SETS OO
VARIABLES oo, attr_i
INVARIANT
    oo \subseteq OO \land
    attr_i \in oo \rightarrow attr_i_type
INITIALISATION oo, attr_i := \{ \}. \{
OPERATIONS
    this \leftarrow new_OO(con_args) \triangleq
        PRE oo \neq OO THEN
            ANY cid WHERE cid \in OO \rightarrow oo THEN
                oo := oo \cup \{ cid \} \land attr_i(cid) := con_i args \land this := cid
        END
    END;
    result \leftarrow method_i(this, args) \triangleq
        PRE this \in oo \land args \in args_type
        THEN body_i includes assignment of result
        END
END
```

1.1 Important parts of model to note

1. The set of all possible objects in the class is modelled by a deferred set, OO. This can be thought of as a set of pointers. **Note carefully:** a machine only models a non-static class if it uses a deferred set to manage a set of objects.

2. The subset of all existing objects is modelled by a variable subset, oo.

3. The non-static attributes will all appear will appear as variables that are total functions from oo to the type of the attribute.

4. The methods are modelled as operations for which there is an extra argument, this, which is the object reference (pointer).

*Note: if the class had static attributes then these would be modelled by variables that were not functional in oo.*
2 From B to UML: A Case Study

We will now consider going in the opposite direction: from a set of B machines to a UML class diagram.

2.1 A Program Manager

For our case study we will model a small —but reasonably realistic— program management system. The entities in this system are:

Courses:  
courses have:

- a name: each course much have a unique name;
- prerequisites: a set of courses that are prerequisites with the following constraints:
  - a course cannot be a prerequisite for itself;
  - as a consequence, there can be no cycles in the prerequisite chains.
- an identity: assigned by the system.

Gened: gened is a distinguished subset of courses.

Programs: programs are collections of courses. The courses are partitioned into:

- Core: a fixed collection of courses, all of which must be completed.
- Electives: a fixed collection of which, from which a fixed number must be completed.
- Gened: a fixed number of Gened courses must be completed.

The sets of courses comprising Core, Electives and Gened must be pairwise disjoint.

Students: students enrol in programs and courses subject to the following constraints:

- enrolment in programs: a student can be enrolled in only one program at a time.
- enrolment courses: a student may be enrolled in courses contained in the program, in which they are enrolled. Having completed any of the core, elective or gened components they may not enrol in any further courses in that component.

Assessors: assessors are accredited to determine completion or failure of courses. In this case study no graded results are considered.

Program managers: program managers determine completion of programs.

2.2 A Note on Modelling

The modelling brings out some problems in the assumptions about program management, such as:

what is program core, electives and gened throughout a student’s enrolment in a program given that each of these can be redefined at any stage? The following model does not resolve this problem. A problem exists because the determination of student “completion” is defined by comparing the courses completed with the current core, elective and gened content of a program. But the current model allows these to change, so a course that was core at the time of enrolling in the course may no longer be core some time later.
Two possible solutions are suggested:

1. a program could be *locked* in which state the program could not be changed. Students would only be allowed to enrol in locked programs. This effectively allows an interval during which a program could be defined before being locked.

2. revision levels could be associated with each program and each revision could be locked. A student would enrol in a particular revision level of a program.

**Formal model of ProgramManager**

The formal model can be found in the appendix.

**A Note on Development**

For reasons to do with the desired structuring of the final implementation, the machines are composed using `EXTENDS`. This does not determine the OO class relationships, that is, `EXTENDS` does not necessarily imply subclassing. Subclassing only occurs when the extended machine imposes extra constraints on the same objects. For example, GenEd is a subclass of Course; Course is not a subclass of Program.

### 2.3 A Class Diagram

![Class Diagram](image-url)

### 2.4 Notes on B Objects

Although entity classes for *Course, Program, Student*, etc have been provided there are a number of points that must be appreciated:
• the classes, and their associated objects, are implemented in a different implementation layer;

• while it is possible to use the classes in a UML model, it is not possible for the class types to be used literally in, say, a Java implementation, although they can be used conceptually via their references;

• object creation and method calling is essentially delegated through ProgramManagerAPI, which is represented by a static class, ProgramManagerAPIJNI in the Java Native Interface (JNI). More information on this will be provided in a later lecture.

3 Implementation

We will present the implementation in two stages:

1. the implementation of ProgramManagerAPI through importing ProgramManager. In preparation for the implementation of the guards in ProgramManagerAPI the component machines have auxiliary operations for evaluating precondition conjuncts.

2. the implementation of ProgramManager using a database machine generated by the B-Toolkit.

3.1 Augmentation of machines

Because of complete hiding in implementation —variables are only visible in invariants, and all reading or writing of variable values must be handled through operations— we need to add extra operations to the machines. In particular, we augment the machines with operations that can be used to determine preconditions.

3.2 Implementing Class Attributes

The attributes of a class can be implemented using a database. Essentially, a record is a data object. For example, CourseDB is a specification of the attributes of the Course/GenEd class.

3.3 Notes on the form of guards

Some precondition conjuncts in the fragile operations have the form $P \Rightarrow Q$. For example in the EnrolCourse operation in Student, we want express the notion, “if the course the student is trying to enrol in is a core course in the program, then the student hasn’t already completed all core courses”. This is easily expressed by implication. When we form a guard in EnrolCourse, the robust version of enrolCourse in ProgramManagerAPI, from the precondition $P \Rightarrow Q$ in enrolCourse, we need $\neg(P \Rightarrow Q)$, but $P \Rightarrow Q \equiv \neg P \lor Q$, and so $\neg(P \Rightarrow Q) \equiv P \land \neg Q$; implication changes to conjunction, and that is more comfortable than disjunction.

3.4 Proof Obligations

Given the development process from ProgramManager to ProgramManagerAPI to ProgramManagerAPII, we might expect very few, or only easily proved, proof obligations from the operations in a machine like ProgramManagerAPII. If that were the case then proof obligation generation will give very strong feedback on the soundness of the development.
For the current revision of the development ProgramManagerAPII has 156 POs of which 96 remain undischarged after the first run of the autoprover. That may mean that there are problems with the new version of ProgramManagerAPII, but this has not yet been investigated.

3.5 Implementation of ProgramManager

The implementation of ProgramManagerAPI imported ProgramManager, so now we have to implement that machine.

We are going to use the Base Generator of the B-Toolkit, and there are essentially two different strategies:

A define a single base for ProgramManager and implement ProgramManager in a single step, or

B define separate bases for each of the component machines (GenEd, Program, etc) and implement each of these machines in turn, importing the predecessor machines.

(B) has the advantage that the operations of component machines can be used to implement the operations of the machines that import those component machines. This can’t be done with (A) as there is only a single machine being implemented. The disadvantage of (B) is that the bases of component machines are hidden and so auxiliary operations need to be added to gain access to information.

It is not clear which of (A) and (B) is the simpler. We choose (B) here.

An example of a single (monolithic) base is shown in the following ProgramManagerDB. Remember this is not used here, but could be used to make a single implementation of the ProgramManager machine. Notice that the component bases contain mutual references. In the distributed base solution that we use here those references are replaced by external names that become machine parameters to the base machines.

3.6 ProgramManagerDB base

```system
SYSTEM ProgramManagerDB
SUPPORTS ProgramManagerI
IS
GLOBAL
  GenEd ∈ SET ( COURSEDB ) [ 100 ]
END ;

BASE COURSEDB
MANDATORY
  CourseName ∈ STRING [ 50 ] ;
  PreReqs ∈ SEQ ( COURSEDB ) [ 100 ] ;
  PreReqClosure ∈ SET ( COURSEDB ) [ 100 ] ;
  remPreReqs ∈ N
END ;

BASE PROGRAMDB
MANDATORY
  ProgName ∈ STRING [ 100 ] ;
  ProgCore ∈ SEQ ( COURSEDB ) [ 100 ] ;
```
3.7 Incremental implementation

The implementation presented here will be done in the following steps:

1. implementation of Course/GenEd using a course database;
2. implementation of Program using a program database and importing GenEdI;
3. implementation Student using a student database and importing ProgramI;
4. implementation of Assessor using an assessor database and importing StudentI;
5. implementation of ProgramManager importing AssessorI. No new base is required in this step since ProgramManager has no variables.

Essentially this implementation proceeds by a series of implementation extension steps. At the implementation level there is no import instruction similar to the extends instruction, so we have to explicitly promote operations of the imported machine.

4 Invariant in an Implementation

The invariant in an implementation relates the variables in the imported machines to the variables in the machine being implemented. The invariant is not necessary for the implementation to be analysed, but it is necessary for the generation of meaningful POs.
Invariants are present in all the implementation machines presented here, and illustrate how the variables in the imported machines are used to implement the variables in the implemented machine. Notice that ProgramManagerAPI and ProgramManager have the same state, so no invariant is required in ProgramManagerAPIII.

5 Invariant and Variant in a while construct

In addition to the body of a while construct, which essentially contains the implementation of the construct being implemented there is an invariant and a variant. For examples see the while constructs in the implementation of Student.

The informal understanding of the invariant is that it is true initially, before entering the while, and no matter how many iterations of the loop it will remain true. It follows that it will also be true when the loop terminates. Of course, when the loop terminates the guard of the loop will be false, so we have

\[ \neg (\text{guard}) \land \text{invariant} \Rightarrow \text{loopGoal} \]

Notice that in the above referenced while constructs the invariant references variable of the machine being implemented, while the body of the while loop is modifying and using local variables and variables of the imported machines. The invariant of the implementation machines provides the “glue” that links these together.

The variant is an expression whose value is a natural number, which is guaranteed to decrease on each iteration. Since the natural numbers have a lower bound of 0, it follows that the loop cannot iterated forever.

So the invariant is used to prove correctness of the loop, and the variant is used to proved termination. The former without the latter is called partial correctness.

6 Using Strings

The STRING type (seq(CHAR)) is defined in the String_TYPE type machine. It should be noted that this machine is seen by all implementation machines.

In order to implement the string type correctly in the intermediate machines a properties clause similar to the following is in included in those machines:

\[ \text{PROPERTIES} \]

\[ \text{COURSENAME} = \text{STRING} \]

which equates the deferred name sets to STRING. The correctness of this clause is in some doubt, but it is necessary to achieve the correct code.

In the base specifications there is a STRING[size] specification, and it should be noted that this is used instead of the deferred set name wherever a string type is required. So, the implementation to string is split between the base specification and the machine implementation. It should also be noted that there are different operations for referencing string fields in a base.

7 Generating JNI

The JNI for an implementation is generated by specifying the JNI interface option and then generating an interface for a machine.
8 A Guide to JNI

The file that make up the JNI for the interface to ProgramManagerAPI can be found in CDE/C/ProgramManagerAPI.

Most of the files in this directory form the JNI to implementation of the ProgramManagerAPI and should not be modified. They form a JNI to any Java implemented interface.

The following is a list of significant files in this directory:

ProgramManagerAPIJNI.java The generated JNI interface. This could be used as the basis of a more elaborate GUI, and with its accompanying dialogue programs would be modified. This also contains the declaration of the public class ProgramManagerAPIJNI that defines the JNI. This part should not be changed.

ProgramManagerAPIJNIImp.c Contains the C-code that forms a bridge between the C translation of the native methods and the C-code of the translation the corresponding B operation. This file should not be changed.

A discussion of what is needed to integrate a Java GUI with the JNI can be found in the document Using Native C Code with Java: The Java Native Interface (JNI).

9 Revision notes

1.5 Small formatting changes and a simple, but elusive implementation bug identified by using ANSI C for translation.

1.4 Features include:

- More iterations across sets.
- Added Save/Restore across all component machines.

1.3 An almost complete version. Some operations are missing, POs have not all been discharged, and the fundamental issue of dynamic programs and program completion has not been resolved.

Features of this version are:

1. This version comes with complete executables for the sub-components: GenEd/Course, Program, Student, Assessor and ProgramManagerAPI.

2. All external (deferred) types have been equated to STRING, so all communication through the ProgramManagerAPI interface are via strings.

3. The JNI interface uses tabs for partitioning the operations into classes: Course, Program, Student, Assessor, ProgramManager

1.2 A number of changes have been made since the last revision, notably the removal of all operations that return sets. For examples, see the checking of prerequisites in the operations prereqsCompleted and coreCompleted in the Student machine. Look at the implementation of Student to see while loops being used to implement subsetting.

The development of the program manager has been substantially revised since the previous version. The following points should be noted:
All operations at the API interface level use names (strings). This seems appropriate to an interface; internal identifiers are unknown to the user of an interface and nor is it appropriate for those identifiers to be known. Once exception is made in this case study: the internal identifier for a student is made known and is used as the unique student identifier.

Almost all operations below the interface do not use names: they use the internal identifiers. This produces dramatic implementations in which there are virtually no guards: the operations proceed on the assumption of preconditions. Those preconditions are checked by guards in the API interface.

The implementation is not yet complete but the strategy should be clear. The next steps would be the implementation of Assessor and ProgramManager.

Use of sets as arguments and results: This development has some instances of sets being returned from and passed to operations: in the implementation of firstPreReq in GenEdI, and the implementation of prereqsCompleted in StudentI. The former should be OK, but the latter is probably not correct as the sets are being passed between two different machines: CourseDB and StudentDB. The BToolkit does not diagnose an error, but there is suspicion that there will be a problem. This could be overcome by using a single database for courses and programs, rather than separate databases.

10 Unfinished Work

Proof obligations have not all been discharged.

No guarantees are given!

A Formal model of Program Manager

The formal model of the Program Manager is presented here as a set of B machines.

A.1 Course machine

The first machines essentially specify a class of Course. Notice that the seen machine CourseName simply establishes a type.

MACHINE CourseName
SETS COURSENAME

OPERATIONS
\[ \text{csname} \leftarrow \text{anyCOURSENAME} \equiv \text{csname} : \in \text{COURSENAME} \]
END

MACHINE Course_CTX
SETS COURSE
END
MACHINE Course (maxCourse, maxPrereqs)

CONSTRAINTS maxCourse ∈ N^1 ∧ maxPrereqs ∈ N^1

SEES CourseName, Course_CTX, Bool_TYPE

VARIABLES
courses, coursename, prereqs, prereqclosure, remprereqs

INVARIANT
courses ⊆ COURSE ∧
card (courses) ≤ maxCourse ∧
coursename ∈ courses ↔ COURSENAME ∧
prereqs ∈ courses ↔ courses ∧
prereqclosure ∈ courses ↔ courses ∧
prereqclosure = prereqs^* ∧
∀ cs . (cs ∈ dom (prereqs) ⇒
  cs ∉ prereqclosure [ { cs } ] ) ∧
remprereqs ⊆ prereqs

ASSERTIONS
prereqs ∩ id (courses) = {} ∧
∀ cs . (cs ∈ dom (prereqs) ⇒
  cs ∉ prereqs^* [ { cs } ] )

INITIALISATION
courses, coursename := {}, {} ||
prereqs, prereqclosure, remprereqs := {}, {}, {}

OPERATIONS
newCourse (csname) ≡
  PRE csname ∈ COURSENAME ∧
  csname ∉ ran (coursename) ∧
  card (courses) ≠ maxCourse
  THEN ANY courseid WHERE courseid ∈ COURSE − courses THEN
  courses := courses ∪ {courseid} ||
  coursename (courseid) := csname
  END
  END;

addPrereq (course, prereq) ≡
  PRE course ∈ COURSE ∧ prereq ∈ COURSE ∧
  course ∈ courses ∧ prereq ∈ courses ∧
  prereq ≠ course ∧
  prereq ∉ prereqs [ { course } ] ∧
  card (prereqs [ { course } ]) ≠ maxPrereqs ∧
  course ∉ prereqs^* [ { prereq } ]
  THEN prereqs := prereqs ∪ {course → prereq} ||
  prereqclosure := prereqclosure ∪
  ({course → prereq} ∪ (prereqclosure ;
  {course} × prereqclosure [ {prereq} ])) ||
\[
\text{remprereqs} := \{ \text{course} \} \triangle \text{remprereqs}
\]

\text{END} ;
\text{delPrereq} ( \text{course}, \text{prereq} ) \triangleq
\text{PRE} \quad \text{course} \in \text{COURSE} \land \text{prereq} \in \text{COURSE} \land \\
\text{course} \in \text{courses} \land \text{prereq} \in \text{courses} \land \\
\text{prereq} \in \text{prereqs} [ \{ \text{course} \} ] \\
\text{THEN} \quad \text{prereqs} := \text{prereqs} - \{ \text{course} \mapsto \text{prereq} \} \triangleq \\
\text{prereqclosure} := \text{prereqclosure} \triangleq ( \text{prereqs} - \{ \text{course} \mapsto \text{prereq} \} )^*
\text{END} ;

The following two operations represent an iterative alternative to an operation \text{allPreReqs} for use at the interface.

\text{ok} , \text{prereq} \leftarrow \text{firstPreReq} ( \text{course} ) \triangleq
\text{PRE} \quad \text{course} \in \text{COURSE} \land \text{course} \in \text{courses}
\text{THEN} \quad \text{IF} \quad \text{course} \in \text{dom} ( \text{prereqs} ) \text{ THEN}
\text{ANY} \quad \text{cs} \text{ WHERE} \quad \text{cs} \in \text{prereqs} [ \{ \text{course} \} ] \text{ THEN}
\text{ok} := \text{TRUE} \parallel \text{prereq} := \text{cs} \parallel \\
\text{remprereqs} := \text{remprereqs} \triangleq ( \text{prereqs} - \{ \text{course} \mapsto \text{prereq} \} )^* \\
\{ \text{course} \mapsto \text{cs} \}
\text{END}
\text{ELSE} \quad \text{ok} := \text{FALSE} \parallel \text{prereq} :\in \text{COURSE} \parallel \\
\text{remprereqs} := \{ \text{course} \} \triangle \text{remprereqs}
\text{END}
\text{END} ;
\text{ok} , \text{prereq} \leftarrow \text{nextPreReq} ( \text{course} ) \triangleq
\text{PRE} \quad \text{course} \in \text{COURSE} \land \text{course} \in \text{courses}
\text{THEN} \quad \text{IF} \quad \text{course} \in \text{dom} ( \text{remprereqs} ) \text{ THEN}
\text{ANY} \quad \text{cs} \text{ WHERE} \quad \text{cs} \in \text{remprereqs} [ \{ \text{course} \} ] \text{ THEN}
\text{ok} := \text{TRUE} \parallel \text{prereq} := \text{cs} \parallel \\
\text{remprereqs} := \text{remprereqs} - \{ \text{course} \mapsto \text{cs} \}
\text{END}
\text{ELSE} \quad \text{ok} := \text{FALSE} \parallel \text{prereq} :\in \text{COURSE}
\text{END}
\text{END} ;

Operations providing preconditions

\text{ok} \leftarrow \text{MoreCourses} \triangleq
\text{BEGIN} \quad \text{ok} := \text{bool} ( \text{card} ( \text{courses} ) \neq \text{maxCourse} ) \text{ END} ;
\text{ok} \leftarrow \text{newCourseName} ( \text{name} ) \triangleq
\text{PRE} \quad \text{name} \in \text{COURSENAME}
\text{THEN} \quad \text{ok} := \text{bool} ( \text{name} \notin \text{ran} ( \text{coursename} ) )
\text{END} ;
\text{ok} , \text{csid} \leftarrow \text{CourseExists} ( \text{csname} ) \triangleq
\text{PRE} \quad \text{csname} \in \text{COURSENAME}
\text{THEN} \quad \text{ok} := \text{bool} ( \text{csname} \notin \text{ran} ( \text{coursename} ) ) \parallel \\
\text{csid} := \text{coursename}^{-1} ( \text{csname} )
A.2 GenEd Subclass machine

Within Course we establish a subclass GenEd.

MACHINE GenEd (maxCourse, maxPrereqs)
CONSTRAINTS maxCourse $\in \mathbb{N}_1$ ∧ maxPrereqs $\in \mathbb{N}_1$
SEES CourseName, Course_CTX, Bool_TYPE
EXTENDS Course (maxCourse, maxPrereqs)
VARIABLES gened
INVARIANT gened $\subseteq$ courses
INITIALISATION gened := {}
PRE course ∈ COURSE ∧ course ∈ courses
THEN ok := bool ( course ∈ gened )
END;
ok ← savegened ≡
BEGIN ok := BOOL || skip END;
ok ← restoregened ≡
ANY ge WHERE ge ⊆ courses
THEN gened := ge || ok := BOOL
END
END

A.3 Program machine

Now we specify a machine that defines a class Program, which contains Course but not as a superclass.

MACHINE ProgName
SETS PROGNAME

OPERATIONS
  prname ← anyPROGNAME ≡
  prname ∈ PROGNAME
END

MACHINE Program_CTX
SETS PROGRAM
END

MACHINE Program ( maxProg , maxCourse , maxPrereqs )
CONSTRAINTS maxProg ∈ N₁ ∧ maxCourse ∈ N₁ ∧ maxPrereqs ∈ N₁
SEES ProgName , Program_CTX , CourseName , Course_CTX , Bool_TYPE
EXTENDS GenEd ( maxCourse , maxPrereqs )
VARIABLES
  programs ,
  progname ,
  progcocore ,
  remprogcore ,
elective ,
  remelective ,
electcount ,
geccount

INVARIANT
  programs ⊆ PROGRAM ∧
  progname ∈ programs ⇔ PROGNAME ∧
  progcocore ∈ programs ⇔ courses ∧
  remprogcore ∈ programs ⇔ courses ∧
elective ∈ programs ↔ courses ∧
remelective ∈ programs ↔ courses ∧
electcount ∈ programs → N ∧
egecount ∈ programs → N ∧
ransite (progcore) ∩ gened = {} ∧
ransite (elective) ∩ gened = {} ∧
∀ pr . (pr ∈ programs ⇒
progcore [ { pr } ] ∩ elective [ { pr } ] = { })

INITIALISATION
programs, progname := {}, {} ||
progcore, remprogcore := {}, {} ||
elective, remelective := {}, {} ||
electcount, gecount := {}, {} ||

OPERATIONS
newProgram (prname) ≜

PRE prname ∈ PROGNAME ∧ card (programs) ≠ maxProg

THEN ANY pid WHERE pid ∈ PROGRAM − programs

THEN programs := programs ∪ { pid } ||

progname (pid) := prname ||

progcore := { pid } ≪ progcore ||

remprogcore := { pid } ≪ remprogcore ||

elective := { pid } ≪ elective ||

electcount (pid) := 0 ||

gecount (pid) := 0

END

END;

addCore (program, course) ≜

PRE program ∈ PROGRAM ∧ course ∈ COURSE ∧
program ∈ programs ∧ course ∈ courses ∧
course ∉ gened ∧ course ∉ elective [ { program } ]

THEN progcore := progcore ∪ { program → course } ||

remprogcore := { program } ≪ remprogcore

END;

delCore (program, course) ≜

PRE program ∈ PROGRAM ∧ course ∈ COURSE ∧
program ∈ programs ∧ course ∈ courses ∧ course ∈ progcore [ { program } ]

THEN . progcore := progcore − { program → course }

END;

addElective (program, course) ≜

PRE program ∈ PROGRAM ∧ course ∈ COURSE ∧
program ∈ programs ∧ course ∈ courses ∧
course ∉ gened ∧ course ∉ progcore [ { program } ]

THEN . elective := elective ∪ { program → course }

END;

delElective (program, course) ≜

PRE program ∈ PROGRAM ∧ course ∈ COURSE ∧
program ∈ programs ∧ course ∈ courses ∧ course ∈ elective [ { program } ]
THEN elective := elective - { program \mapsto course }

END;

setElectCount (program, count) \equiv

\text{PRE} \quad \text{program} \in \text{PROGRAM} \land count \in \mathbb{N} \land \text{program} \in \text{programs}

THEN electcount (program) := count

END;

setGECount (program, count) \equiv

\text{PRE} \quad \text{program} \in \text{PROGRAM} \land count \in \mathbb{N} \land \text{program} \in \text{programs}

THEN gecount (program) := count

END;

Operations providing information

prname \leftarrow progName (program) \equiv

\text{PRE} \quad \text{program} \in \text{PROGRAM} \land \text{program} \in \text{programs}

THEN prname := progname (program)

END;

Operations providing preconditions

ok \leftarrow MorePrograms \equiv

\text{BEGIN} \quad ok := \text{bool ( card (programs) \neq maxProg )} \text{ END };

ok \leftarrow newProgName (prname) \equiv

\text{PRE} \quad prname \in \text{PROGRAM} \text{ THEN } ok := \text{bool ( prname \notin ran ( progname ) )} \text{ END };

ok, program \leftarrow ProgramExists (prname) \equiv

\text{PRE} \quad prname \in \text{PROGRAM}

THEN

ok := \text{bool ( prname \in ran ( progname ) )} \lor

\text{program} := \text{progname}^{-1} (prname)

END;

ok \leftarrow isCore (program, course) \equiv

\text{PRE} \quad \text{program} \in \text{PROGRAM} \land \text{course} \in \text{COURSE}

THEN ok := \text{bool ( course \in progcore [ \{ program \} ] )}

END;

ok \leftarrow isElective (program, course) \equiv

\text{PRE} \quad \text{program} \in \text{PROGRAM} \land \text{course} \in \text{COURSE}

THEN ok := \text{bool ( course \in elective [ \{ program \} ] )}

END;

ok, course \leftarrow firstCore (program) \equiv

\text{PRE} \quad \text{program} \in \text{PROGRAM} \land \text{program} \in \text{programs}

THEN IF \text{ program } \in \text{dom ( progcore ) } \text{ THEN}

ANY cs \text{ WHERE } cs \in \text{progcore [ \{ program \} ]}

THEN ok := \text{TRUE} \lor course := cs \lor

remprogcore := remprogcore \triangleq (\text{ program } \triangleq progcore) -

\{ \text{ program } \mapsto cs \}

END

ELSE ok := \text{FALSE} \lor course \in \text{COURSE} \lor

remprogcore := \{ \text{ program } \triangleq \text{remprogcore}

END

17
END;
ok, course ← nextCore (program) ≜
  PRE program ∈ PROGRAM ∧ program ∈ programs
  THEN IF program ∈ dom (remprogcore) THEN
    ANY cs WHERE cs ∈ remprogcore [ { program } ]
    THEN ok := TRUE || course := cs ||
    remprogcore := remprogcore − { program → cs }
  END
  ELSE ok := FALSE || course ∈ COURSE
  END
END;
ok, course ← firstElective (program) ≜
  PRE program ∈ PROGRAM ∧ program ∈ programs
  THEN IF program ∈ dom (elective) THEN
    ANY cs WHERE cs ∈ elective [ { program } ]
    THEN ok := TRUE || course := cs ||
    remelective := remelective ≤ ( { program } ≤ elective ) −
    { program → cs }
  END
  ELSE ok := FALSE || course ∈ COURSE ||
    remelective := { program } ≤ remelective
  END
END;
ok, course ← nextElective (program) ≜
  PRE program ∈ PROGRAM ∧ program ∈ programs
  THEN IF program ∈ dom (remelective) THEN
    ANY cs WHERE cs ∈ remelective [ { program } ]
    THEN ok := TRUE || course := cs ||
    remelective := remelective ≤ { program → cs }
  END
  ELSE ok := FALSE || course ∈ COURSE
  END
END;
count ← progElectCount (program) ≜
  PRE program ∈ PROGRAM ∧ program ∈ programs
  THEN count := electcount (program)
END;
count ← progGECCount (program) ≜
  PRE program ∈ PROGRAM ∧ program ∈ programs
  THEN count := gecount (program)
END;
ok ← saveprogram ≜
BEGIN ok := BOOL || skip END;
ok ← restoreprogram ≜
BEGIN
  ANY prg, prn, prv, rpc, pe, rel, ec, gec WHERE
  prg ⊆ PROGRAM ∧
  prn ∈ prg ⇔ PROGNAME ∧
prc ∈ prg ↔ courses ∧
rpc ∈ prg ↔ courses ∧
pe ∈ prg ↔ courses ∧
rel ∈ prg ↔ courses ∧
ec ∈ prg → N ∧
gec ∈ prg → N ∧
ran ( prc ) ∩ gened = { } ∧
ran ( pe ) ∩ gened = { } ∧
∀ pr . ( pr ∈ prg ⇒
pr [ { pr } ] ∩ pe [ { pr } ] = { } )
THEN
programs , progname := prg , prn ||
progcore , remprogcore := prc , rpc ||
elective , remelective := pe , rel ||
electcount , gecount := ec , gec
END || ok :∈ BOOL
END

A.4 Student machine

We now present a machine that represents a Student that is associated with Program.

MACHINE Student_CTX
SETS STUDENT
END

MACHINE Student ( maxStudent , maxProg , maxCourse , maxPrereqs )
CONSTRAINTS
maxStudent ∈ N₁ ∧ maxProg ∈ N₁ ∧
maxCourse ∈ N₁ ∧ maxPrereqs ∈ N₁
SEES
ProgName , Program_CTX , CourseName , Course_CTX ,
Student_CTX , Bool_TYPE
EXTENDS Program ( maxProg , maxCourse , maxPrereqs )
VARIABLES
students ,
progenrolled ,
progcompleted ,
courseenrolled ,
coursecompleted ,
coursefailed ,
electivecompleted ,
genedcompleted
INVARIANT
students ⊆ STUDENT ∧
card ( students ) ≤ maxStudent ∧
progenrolled ∈ students → programs ∧
progcompleted ∈ students → programs ∧
courseenrolled ∈ students → courses ∧
coursecompleted ∈ students → courses ∧
electivecompleted ∈ students → ℕ ∧
genedcompleted ∈ students → ℕ ∧
∀ st. ( st ∈ dom ( gecount ( progenrolled ( st ) ) ) ) ∧
∀ st. ( st ∈ dom ( electivecompleted ) ) ⇒
egenericcompleted ( st ) ≤ gecount ( progenrolled ( st ) ) ∧
electivecompleted ( st ) ≤ electcount ( progenrolled ( st ) ) ∧
coursefailed ∈ students → courses ∧
progenrolled ∩ progcompleted = {} ∧
courseenrolled ∩ coursecompleted = {}

More consistency constraints needed

INITIALISATION
students , progenrolled , progcompleted := {} , {} , {} ||
courseenrolled , coursecompleted , coursefailed := {} , {} , {} ||
electivecompleted , gnedcompleted := {} , {} ||

OPERATIONS

\begin{align*}
\text{student} & \leftarrow \text{newStudent} \triangleq \\
\text{PRE} & \quad \text{card (students)} \not= \text{maxStudent} \\
\text{THEN} & \quad \text{ANY st WHERE st ∈ STUDENT} - \text{students} \\
\text{THEN} & \quad \text{students} := \text{students} \cup \{ st \} || \\
\text{student} & := st \\
\text{END} ;
\end{align*}

\begin{align*}
enrolProg & \ ( \text{student} , \text{program} ) \triangleq \\
\text{PRE} & \quad \text{student} \in \text{STUDENT} \land \text{program} \in \text{PROGRAM} \land \\
& \quad \text{student} \in \text{students} \land \text{student} \not\in \text{dom (progenrolled)} \land \\
& \quad \text{program} \in \text{programs} \\
\text{THEN} & \quad \text{progenrolled (student)} := \text{program} || \\
& \quad \text{electivecompleted (student)} := 0 || \\
& \quad \text{genedcompleted (student)} := 0 \\
\text{END} ;
\end{align*}

\begin{align*}
enrolCourse & \ ( \text{student} , \text{course} ) \triangleq \\
\text{PRE} & \quad \text{student} \in \text{STUDENT} \land \text{course} \in \text{COURSE} \land \\
& \quad \text{student} \in \text{dom (progenrolled)} \land \\
& \quad \text{course} \in \text{courses} \land \\
& \quad \text{course} \not\in \text{courseenrolled ( student ) } \land \\
& \quad \text{course} \not\in \text{coursecompleted ( student ) } \land \\
& \quad \text{prereqs ( course ) } \subseteq \text{coursecompleted ( student ) } \land \\
& \quad \text{course} \in \text{progcore ( progenrolled ( student ) ) } \cup \\
& \quad \text{( elective [ progenrolled ( student ) ] } \cup \text{ gned ) } \land \\
& \quad \text{( course ∈ progcore ( progenrolled ( student ) ) ) } \Rightarrow \\
& \quad \neg ( \text{progcore ( progenrolled ( student ) ) } \subseteq \\
& \quad \text{coursecompleted ( student ) } ) \land \\
\end{align*}
( course ∈ elective [ { progenrolled ( student ) } ] ) ⇒
  card ( coursecompleted [ { student } ] ∩
  elective [ { progenrolled ( student ) } ] ) ≠
  electcount ( progenrolled ( student ) ) ) ∧

( course ∈ gened ⇒
  card ( coursecompleted [ { student } ] ∩ gened ) ≠
  gecount ( progenrolled ( student ) ) )

THEN  courseenrolled := courseenrolled ∪ { student ↔ course }

END ;
disenrolCourse ( student , course ) ≡
  PRE  student ∈ STUDENT ∧ course ∈ COURSE ∧
  student ∈ dom ( progenrolled ) ∧
  course ∈ courseenrolled [ { student } ]

THEN  courseenrolled := courseenrolled − { student ↔ course }

END ;
completeCourse0 ( student , course ) ≡
  PRE  student ∈ STUDENT ∧ course ∈ COURSE ∧
  student ∈ students ∧ course ∈ courseenrolled [ { student } ]

THEN  coursecompleted := coursecompleted ∪ { student ↔ course } ||
  courseenrolled := courseenrolled − { student ↔ course } ||

SELECT  course ∈ elective [ { progenrolled ( student ) } ] THEN
  electivecompleted ( student ) := electivecompleted ( student ) + 1

WHEN  course ∈ gened THEN
  genedcompleted ( student ) := genedcompleted ( student ) + 1

END

END ;
failCourse0 ( student , course ) ≡
  PRE  student ∈ STUDENT ∧ course ∈ COURSE ∧
  student ∈ students ∧ course ∈ courseenrolled [ { student } ]

THEN  coursefailed := coursefailed ∪ { student ↔ course } ||
  courseenrolled := courseenrolled − { student ↔ course } ||

END ;
completeProg0 ( student , program ) ≡
  PRE  student ∈ STUDENT ∧ program ∈ PROGRAM ∧
  student ∈ dom ( progenrolled ) ∧ program ∈ programs ∧
  progenrolled ( student ) = program ∧
  student ∈ dom ( courseenrolled )

THEN  progcompleted := progcompleted ∪ { student ↔ program } ||
  progenrolled := { student } ≡ progenrolled

END ;

Operations providing preconditions

ok ← moreStudents ≡
  BEGIN  ok := bool ( card ( students ) ≠ maxStudent ) END ;

ok ← validStudent ( student ) ≡
  PRE  student ∈ STUDENT THEN  ok := bool ( student ∈ students ) END ;

ok ← studentEnrolled ( student ) ≡
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{student} \in \text{students} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{student} \in \text{dom} ( \text{progenrolled} ) ) \\
\text{END} ;
ok \leftarrow \text{inProgram} ( \text{student} , \text{course} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{course} \in \text{COURSE} \land \\
\quad \text{student} \in \text{students} \land \text{course} \in \text{courses} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{course} \in \text{progcore} \left[ \{ \text{progenrolled} ( \text{student} ) \} \right] \cup \\
\quad \quad ( \text{elective} \left[ \{ \text{progenrolled} ( \text{student} ) \} \right] \cup \text{gened} ) ) \\
\text{END} ;
ok \leftarrow \text{courseEnrolled} ( \text{student} , \text{course} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{course} \in \text{COURSE} \land \\
\quad \text{student} \in \text{students} \land \text{course} \in \text{courses} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{course} \in \text{courseenrolled} \left[ \{ \text{student} \} \right] ) \\
\text{END} ;
ok \leftarrow \text{prereqsCompleted} ( \text{student} , \text{course} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{course} \in \text{COURSE} \land \\
\quad \text{student} \in \text{students} \land \text{course} \in \text{courses} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{prereqs} \left[ \{ \text{course} \} \right] \subseteq \text{coursecompleted} \left[ \{ \text{student} \} \right] ) \\
\text{END} ;
ok \leftarrow \text{coreCompleted} ( \text{student} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{student} \in \text{students} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{progcore} \left[ \{ \text{progenrolled} ( \text{student} ) \} \right] \subseteq \\
\quad \text{coursecompleted} \left[ \{ \text{student} \} \right] ) \\
\text{END} ;
ok \leftarrow \text{electiveCompleted} ( \text{student} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{student} \in \text{students} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{card} ( \text{coursecompleted} \left[ \{ \text{student} \} \right] \cap \\
\quad \text{elective} \left[ \{ \text{progenrolled} ( \text{student} ) \} \right] ) = \\
\quad \text{electcount} ( \text{progenrolled} ( \text{student} ) ) ) \\
\text{END} ;
ok \leftarrow \text{genedCompleted} ( \text{student} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{student} \in \text{students} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{card} ( \text{coursecompleted} \left[ \{ \text{student} \} \right] \cap \text{gened} ) = \\
\quad \text{gecount} ( \text{progenrolled} ( \text{student} ) ) ) \\
\text{END} ;
ok \leftarrow \text{coursesEnrolled} ( \text{student} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{student} \in \text{students} \\
\text{THEN} \quad \text{ok} := \text{bool} ( \text{student} \in \text{dom} ( \text{courseenrolled} ) ) \\
\text{END} ;
\text{program} \leftarrow \text{programEnrolled} ( \text{student} ) \equiv
\text{PRE} \quad \text{student} \in \text{STUDENT} \land \text{student} \in \text{dom} ( \text{progenrolled} ) \\
\text{THEN} \quad \text{program} := \text{progenrolled} ( \text{student} ) \\
\text{END} ;
student ← anyStudent ≜
BEGIN
  student :∈ STUDENT
END;
ok ← savestudent ≜
BEGIN ok :∈ BOOL || skip END;
ok ← restorestudent ≜
BEGIN
  ANY st, pe, pc, ce, cc, cf, ec, gc WHERE
  st ∈ STUDENT ∧
  card(st) ≤ maxStudent ∧
  pe ∈ st ↔ programs ∧
  pc ∈ st ↔ programs ∧
  ce ∈ st ↔ courses ∧
  cc ∈ st ↔ courses ∧
  ec ∈ st → N ∧
  gc ∈ st → N ∧
  ∀ st. (st ∈ dom(gc) ⇒
    gc(st) ≤ gecount(pe(st)) ∧
  ∀ st. (st ∈ dom(ec) ⇒
    ec(st) ≤ electcount(pe(st)) ∧
  cf ∈ st ↔ courses ∧
  pe ∩ pc = {} ∧
  ce ∩ cc = {} ∧
  THEN
  students, progenrolled, progcompleted := st, pe, pc ||
  courseenrolled, coursecompleted, coursefailed := ce, cc, cf ||
  electivecompleted, genedcompleted := ec, gc
END || ok :∈ BOOL
END

A.5 Assessor machine

An Assessor class is now specified. This class has an aggregate association with Student.

MACHINE AssessorName
SETS ASSESSORNAME

OPERATIONS
  prname ← anyASSESSORNAME ≜
    prname :∈ ASSESSORNAME
END

MACHINE Assessor_CTX
SETS ASSESSOR
CONSTANTS nullassessor
PROPERTIES nullassessor ∈ ASSESSOR
END

MACHINE Assessor ( maxAssessor, maxStudent, maxProg, maxCourse, maxPrereqs )
CONSTRAINTS
maxAssessor ∈ N₁ ∧ maxStudent ∈ N₁ ∧
maxProg ∈ N₁ ∧ maxCourse ∈ N₁ ∧ maxPrereqs ∈ N₁
SEES
ProgName, Program_CTX, CourseName, Course_CTX,
Student_CTX, AssessorName, Assessor_CTX, Bool_TYPE
EXTENDS Student ( maxStudent, maxProg, maxCourse, maxPrereqs )
VARIABLES
assessors, assessorname, assesses
INVARIANT
assessors ⊆ ASSESSOR ∧
card ( assessors ) ≤ maxAssessor ∧
assessorname ∈ assessors ⇔ ASSESSORNAME ∧
assesses ∈ assessors ⇔ courses
INITIALISATION assessors, assesses, assessorname := {}, {}, {}

OPERATIONS
newAssessor ( asname ) ≜
PRE asname ∈ ASSESSORNAME ∧ card ( assessors ) ≠ maxAssessor
THEN ANY aid WHERE aid ∈ ASSESSOR − assessors
THEN assessors := assessors ∪ { aid } ∥
assessorname ( aid ) := asname
END
END;
addAssessor ( assessor, course ) ≜
PRE assessor ∈ ASSESSOR ∧ course ∈ COURSE ∧
assessor ∈ assessors ∧ course ∈ courses
THEN assesses := assesses ∪ { assessor ⇔ course }
END;
delAssessor ( assessor, course ) ≜
PRE assessor ∈ ASSESSOR ∧ course ∈ COURSE ∧
assessor ∈ assessors ∧ course ∈ assesses [ { assessor } ]
THEN assesses := assesses − { assessor ⇔ course }
END;
completeCourse ( assessor, student, course ) ≜
PRE assessor ∈ ASSESSOR ∧ student ∈ STUDENT ∧ course ∈ COURSE ∧
assessor ∈ assessors ∧ student ∈ students ∧
course ∈ assesses [ { assessor } ] ∧ course ∈ courseenrolled [ { student } ]
THEN completeCourse0 ( student, course )
END;
failcourse ( assessor, student, course ) ≜
PRE assessor ∈ ASSESSOR ∧ student ∈ STUDENT ∧ course ∈ COURSE ∧
assessor ∈ assessors ∧ course ∈ assesses [ { assessor } ] ∧
student ∈ students ∧ course ∈ courseenrolled [ { student } ]
THEN failCourse0 ( student , course )
END ;

Operations providing preconditions

ok ← MoreAssessors ≡
BEGIN ok := bool ( card ( assessors ) ≠ maxAssessor ) END ;
ok ← newAssessorName ( name ) ≡
PRE name ∈ ASSESSORNAME THEN
  ok := bool ( name ∉ ran ( assessornames ) )
END ;
ok , assessor ← validAssessor ( asname ) ≡
PRE asname ∈ ASSESSORNAME
THEN ok := bool ( asname ∈ ran ( assessornames ) ) ||
  assessor := assessornames⁻¹ ( asname )
END ;
ok ← AssessorAccredited ( assessor , course ) ≡
PRE assessor ∈ ASSESSOR ∧ course ∈ COURSE
THEN ok := bool ( course ∈ assesses [ { assessor } ] )
END ;
ok ← saveassessor ≡
BEGIN ok := BOOL || skip END ;
ok ← restoreassessor ≡
BEGIN
  ANY as , an , ass WHERE
  as ⊆ ASSESSOR ∧
  card ( as ) ≤ maxAssessor ∧
  an ∈ as →→ ASSESSORNAME ∧
  ass ∈ as →→ courses
  THEN assessors , assessornames , assesses := as , an , ass
END || ok := BOOL
END

A.6 Program Manager machine

A ProgramManager machine contains an operation for assessing completion of program. This machine corresponds to a static class that has no attributes, only methods.

MACHINE ProgramManager ( maxAssessor , maxStudent , maxProg ,
  maxCourse , maxPrereqs )
CONSTRAINTS
  maxAssessor ∈ N₁ ∧ maxStudent ∈ N₁ ∧
  maxProg ∈ N₁ ∧ maxCourse ∈ N₁ ∧ maxPrereqs ∈ N₁
SEES
  AssessorName , Assessor_CTX ,
A.7 The Robust Interface

And finally, a machine ProgramManagerAPI that contains only robust operations. This machine corresponds to a top-level, boundary class for this system. It will be a static class with only methods.

MACHINE ProgramManagerAPI (maxAssessor, maxStudent, maxProg,
  maxCourse, maxPrereqs)
CONSTRAINTS
  maxAssessor ∈ N₁ ∧ maxStudent ∈ N₁ ∧
  maxProg ∈ N₁ ∧ maxCourse ∈ N₁ ∧ maxPrereqs ∈ N₁
SEES
  CourseName, Course_CTX,
  ProgName, Program_CTX,
  Student_CTX, Assessor_CTX, AssessorName,
  Bool_CTX
INCLUDES ProgramManager (maxAssessor, maxStudent, maxProg,
  maxCourse, maxPrereqs)
SETS RESPONSE = {OK,
  course_limit_reached,
  program_limit_reached,
  student_limit_reached,
  assessor_limit_reached,
existing_course_name,
existing_program_name,
existing_assessor_name,
no_such_course,
not_gened,
already_gened,
non_existing_prereq,
reflexive_prereq,
cyclic_prereq,
already_prereq,
not_a_prereq,
no_prereqs,
no_more_prereqs,
course_in_gened,
course_in_core,
course_in_elective,
course_not_core,
course_not_elective,
course_not_in_program,
missing_prereqs,
invalid_program,
invalid_course,
invalid_student,
invalid_assessor,
ot_enrolled_student,
ot_enrolled_program,
ot_enrolled_course,
courses_enrolled,
already_enrolled,
course_completed,
core_completed,
electives_completed,
gened_completed,
program_not_completed,
ot_authorised,
save_failed,
restore_failed

OPERATIONS

resp ← NewCourse (csname) ≜

PRE csname ∈ COURSENAME THEN

SELECT card (courses) = maxCourse THEN
    resp := course_limit_reached

WHEN csname ∈ ran (courses) THEN
    resp := existing_course_name

ELSE resp := OK || newCourse (csname)
END
END;
resp := NewgeCourse ( csname ) ⇐
PRE csname ∈ COURSENAME THEN
SELECT csname ∉ ran ( course ) THEN
resp := no_such_course
ELSE ANY course
WHERE course = course ^−1 ( csname ) THEN
SELECT course ∈ gened THEN
resp := already_gened
ELSE resp := OK || newgeCourse ( course )
END
END
END END;

resp := Delgecourse ( csname ) ⇐
PRE csname ∈ COURSENAME THEN
SELECT csname ∉ ran ( course ) THEN
resp := no_such Course
ELSE ANY course
WHERE course = course ^−1 ( csname ) THEN
SELECT course ∉ gened THEN
resp := not_gened
ELSE resp := OK || delgecourse ( course )
END
END
END END;

resp := AddPrereq ( csname , prename ) ⇐
PRE csname ∈ COURSENAME ∧ prename ∈ COURSENAME THEN
SELECT csname ∉ ran ( course ) THEN
resp := no_such_course
WHERE prename ∉ ran ( course ) THEN
resp := non_existing_prereq
ELSE ANY course , prereq
WHERE course = course ^−1 ( csname ) ∧ prereq = course ^−1 ( prename ) THEN
SELECT prereq = course THEN
resp := reflexive_prereq
WHEN prereq ∈ prereqs [ course ] THEN
resp := already_prereq
WHEN course ∈ prereqs * [ prereq ] THEN
resp := cyclic_prereq
ELSE resp := OK || addPrereq ( course , prereq )
END
END
END
END
END ;

28
resp ← DelPrereq (csname, prename) ≡

PRE csname ∈ COURSENAME ∧ prename ∈ COURSENAME THEN
SELECT csname \notin \text{ran}(course) THEN
    resp := no_such_course
WHEN prename \notin \text{ran}(course) THEN
    resp := non_existing_prereq
ELSE ANY course, prereq
    WHERE course = \text{coursename}^{-1}(csname) ∧
        prereq = \text{coursename}^{-1}(prename) THEN
    SELECT prereq = course THEN
        resp := reflexive_prereq
    WHEN prereq \notin prereqs [\{ course \}] THEN
        resp := not_in_prereqs
        ELSE resp := OK || delPrereq (course, prereq)
    END
    END
END
END

END;

FirstPreReq(csname) is specified non-deterministically, but it will be implemented using the implementation
of the firstPreReq operation. Similarly for NextPreReq(csname).

resp , prename ← FirstPreReq (csname) ≡

PRE csname ∈ COURSENAME THEN
SELECT csname \notin \text{ran}(course) THEN
    resp := invalid_course || prename ∈ COURSENAME
ELSE ANY course
    WHERE course = \text{coursename}^{-1}(csname) THEN
    SELECT course \notin \text{dom}(prereqs) THEN
        resp := no_prereqs ||
        prename ∈ COURSENAME
    ELSE resp := OK ||
        prename ∈ \text{coursename} [prereqs [\{ course \}]]
    END
    END
END
END

resp , prename ← NextPreReq (csname) ≡

PRE csname ∈ COURSENAME THEN
SELECT csname \notin \text{ran}(course) THEN
    resp := invalid_course || prename ∈ COURSENAME
ELSE CHOICE resp := OK ||
    prename ∈ \text{coursename} [prereqs [\{ course^{-1} (csname) \}]]
    OR
        resp := no_more_prereqs || prename ∈ COURSENAME
    END
END
END;
resp ← NewProgram ( prname ) =
PRE prname ∈ PROGNAME THEN
SELECT card ( programs ) = maxProg THEN
  resp := program_limit_reached
WHEN prname ∈ ran ( programe ) THEN
  resp := existing_program_name
ELSE resp := OK || newProgram ( prname )
END
END;
resp ← AddCore ( prname , csname ) =
PRE prname ∈ PROGNAME ∧ csname ∈ COURSENAME THEN
SELECT prname ∈ ran ( programe ) THEN
  resp := invalid_program
WHEN csname ∈ ran ( coursename ) THEN
  resp := invalid_course
ELSE ANY program , course WHERE
  program = programe ^−1 ( prname ) ∧
  course = coursename ^−1 ( csname ) THEN
SELECT course ∈ gened THEN
  resp := course_in_gened
WHEN course ∈ elective [ { program } ] THEN
  resp := course_in_elective
ELSE resp := OK || addCore ( program , course )
END
END
END
END;
resp ← DelCore ( prname , csname ) =
PRE prname ∈ PROGNAME ∧ csname ∈ COURSENAME THEN
SELECT prname ∈ ran ( programe ) THEN
  resp := invalid_program
WHEN csname ∈ ran ( coursename ) THEN
  resp := invalid_course
ELSE ANY program , course WHERE
  program = programe ^−1 ( prname ) ∧
  course = coursename ^−1 ( csname ) THEN
SELECT course ∈ progcore [ { program } ] THEN
  resp := course_not_core
ELSE resp := OK || delCore ( program , course )
END
END
END
END;
resp ← AddElective ( prname , csname ) =
PRE prname ∈ PROGNAME ∧ csname ∈ COURSENAME THEN
SELECT prname ∈ ran ( programe ) THEN
resp := invalid_program
WHEN csname \notin ran ( coursename ) THEN
resp := invalid_course
ELSE ANY program, course WHERE
    program = programe^-1 ( prname ) \land
    course = coursename^-1 ( csname ) THEN
SELECT course \in gened THEN
    resp := course_in_gened
WHEN course \in progcore [{ { program } } ] THEN
    resp := course_in_core
ELSE resp := OK || addElective ( program, course )
END
END
END ;
resp := DelElective ( prname, csname ) \equiv
PRE prname \in PROGNAME \land csname \in COURSENAME THEN
SELECT prname \notin ran ( programe ) THEN
resp := invalid_program
WHEN csname \notin ran ( coursename ) THEN
resp := invalid_course
ELSE ANY program, course WHERE
    program = programe^-1 ( prname ) \land
    course = coursename^-1 ( csname ) THEN
SELECT course \notin elective [{ { program } } ] THEN
    resp := course_not_elective
ELSE resp := OK || delElective ( program, course )
END
END
END ;
resp := SetElectCount ( prname, count ) \equiv
PRE prname \in PROGNAME \land count \in \mathbb{N} THEN
SELECT prname \notin ran ( programe ) THEN
resp := invalid_program
ELSE ANY program
    WHERE program = programe^-1 ( prname ) THEN
    resp := OK || setElectCount ( program, count )
END
END
END ;
resp := SetGECount ( prname, count ) \equiv
PRE prname \in PROGNAME \land count \in \mathbb{N} THEN
SELECT prname \notin ran ( programe ) THEN
resp := invalid_program
ELSE ANY program
    WHERE program = programe^-1 ( prname ) THEN

resp := OK \parallel \text{setGEcount}\left(\text{program},\,\text{count}\right) \quad \text{END} \\
\text{END} \\
\text{END} \\
resp,\,\text{sid} \leftarrow \text{NewStudent} \triangleq \begin{aligned} \text{SELECT}\ &\text{card}\left(\text{students}\right) = \text{maxStudent}\ \text{THEN} \\
&\text{resp} := \text{student\_limit\_reached}\ \parallel\ \text{sid} \in \text{STUDENT} \\
\text{ELSE} &\text{resp} := \text{OK}\ \parallel\ \text{sid} \leftarrow \text{newStudent} \\
\text{END} \end{aligned} \\
\text{resp} \leftarrow \text{EnrolProg}\left(\text{sid},\,\text{prname}\right) \triangleq \\
\text{PRE} \ \text{sid} \in \text{STUDENT}\ \land\ \text{prname} \in \text{PROGNAME}\ \text{THEN} \\
\text{SELECT} \ \text{prname} \notin \text{ran}\left(\text{programe}\right)\ \text{THEN} \\
&\text{resp} := \text{invalid\_programe} \\
\text{ELSE} &\text{ANY} \ \text{program} \\
&\text{WHERE} \ \text{program} = \text{programe}^{-1}\left(\text{prname}\right)\ \text{THEN} \\
&\text{SELECT} \ \text{sid} \notin \text{students}\ \text{THEN} \\
&\text{resp} := \text{invalid\_student} \\
&\text{WHEN} \ \text{side} \in \text{dom}\left(\text{progenrolled}\right)\ \text{THEN} \\
&\text{resp} := \text{already\_enrolled} \\
&\text{ELSE} \ \text{resp} := \text{OK}\ \parallel\ \text{enrolProg}\left(\text{sid},\,\text{program}\right) \\
\text{END} \\
\text{END} \\
\text{END} \\
\text{END} \\
\text{resp} \leftarrow \text{EnrolCourse}\left(\text{sid},\,\text{csname}\right) \triangleq \\
\text{PRE} \ \text{sid} \in \text{STUDENT}\ \land\ \text{csname} \in \text{COURSENAME}\ \text{THEN} \\
\text{SELECT} \ \text{csname} \notin \text{ran}\left(\text{coursename}\right)\ \text{THEN} \\
&\text{resp} := \text{invalid\_coursename} \\
\text{ELSE} &\text{ANY} \ \text{course} \\
&\text{WHERE} \ \text{course} = \text{coursename}^{-1}\left(\text{csname}\right)\ \text{THEN} \\
&\text{SELECT} \ \text{sid} \notin \text{dom}\left(\text{progenrolled}\right)\ \text{THEN} \\
&\text{resp} := \text{not\_enrolled\_student} \\
&\text{WHEN} \ \text{course} \in \text{courseenrolled}\left[\{\text{sid}\}\right]\ \text{THEN} \\
&\text{resp} := \text{already\_enrolled} \\
&\text{WHEN} \ \text{course} \in \text{coursecompleted}\left[\{\text{sid}\}\right]\ \text{THEN} \\
&\text{resp} := \text{course\_completed} \\
&\text{WHEN} \ \neg\left(\text{prereqs}\left[\{\text{course}\}\right] \subseteq \text{coursecompleted}\left[\{\text{sid}\}\right]\right)\ \text{THEN} \\
&\text{resp} := \text{missing\_prereqs} \\
&\text{WHEN} \ \text{course} \notin \ \\
&\left(\text{progcource}\left[\{\text{progenrolled}\left(\text{sid}\right)\}\right] \cup \text{elective}\left[\{\text{progenrolled}\left(\text{sid}\right)\}\right] \cup \text{gened}\right)\ \text{THEN} \ \text{resp} := \text{course\_not\_in\_program} \\
&\text{WHEN} \ \text{course} \in \text{progcource}\left[\{\text{progenrolled}\left(\text{sid}\right)\}\right] \wedge \\
&\text{progcource}\left[\{\text{progenrolled}\left(\text{sid}\right)\}\right] \subseteq \text{coursecompleted}\left[\{\text{sid}\}\right]\ \text{THEN} \ \text{resp} := \text{core\_completed} \\
&\text{WHEN} \ \text{course} \in \text{elective}\left[\{\text{progenrolled}\left(\text{sid}\right)\}\right] \wedge \\
&\text{card}\left(\text{coursecompleted}\left[\{\text{sid}\}\right] \cap \text{elective}\left[\{\text{progenrolled}\left(\text{sid}\right)\}\right]\right) = \text{electcount}\left(\text{progenrolled}\left(\text{sid}\right)\right)\ \text{THEN} \\
&\text{resp} := \text{electives\_completed} \\
\text{32}
WHEN course ∈ gened ∧
  card ( coursecompleted [ { sid } ] ∩ gened ) =
    gecount ( progenrolled ( sid ) ) THEN
    resp := gened_completed
ELSE resp := OK || enrolCourse ( sid, course )
END
END

END ;

resp ← DiseinrolCourse ( sid, csname ) ≜

PRE sid ∈ STUDENT ∧ csname ∈ COURSENAME THEN
  SELECT sid ∉ dom ( progenrolled ) THEN resp := not_enrolled_student
ELSE ANY course WHERE course = courserename⁻¹ ( csname ) THEN
  SELECT csname ∈ ran ( courserename ) ⇒
    course ∉ courseenrolled [ { sid } ] THEN resp := not_enrolled_course
  ELSE resp := OK || disenrolCourse ( sid, course )
END
END
END

resp ← NewAssessor ( asname ) ≜

PRE asname ∈ ASSESSORNAME THEN
  SELECT card ( assessors ) = maxAssessor THEN
    resp := assessor_limit_reached
  WHEN asname ∈ ran ( assestorername ) THEN
    resp := existing_assessor_name
  ELSE resp := OK || newAssessor ( asname )
END
END ;

resp ← AddAssessor ( asname, csname ) ≜

PRE asname ∈ ASSESSORNAME ∧ csname ∈ COURSENAME THEN
  SELECT asname ∉ ran ( assestorername ) THEN
    resp := invalid_assessor
  WHEN csname ∉ ran ( courserename ) THEN
    resp := invalid_course
  ELSE ANY aid, course WHERE aid = assessorname⁻¹ ( asname ) ∧
    course = courserename⁻¹ ( csname ) THEN
    resp := OK || addAssessor ( aid, course )
END
END

END ;

resp ← DelAssessor ( asname, csname ) ≜

PRE asname ∈ ASSESSORNAME ∧ csname ∈ COURSENAME THEN
  SELECT asname ∉ ran ( assestorername ) THEN
    resp := invalid_assessor
  WHEN csname ∉ ran ( courserename ) THEN
    resp := invalid_course

33
ELSE ANY aid, course WHERE aid = assessorname^{-1} (asname) \land
course = coursename^{-1} (csname) THEN
resp := OK \parallel delAssessor (aid, course)
END
END
END;
resp ← CompleteCourse (asname, sid, csname) ∨
PRE asname ∈ ASSESSORNAME \land sid ∈ STUDENT \land
cid ∈ students \land csname ∈ COURSENAME THEN
SELECT asname /∈ ran (assessorname) THEN
resp := invalid_assessor
WHEN csname /∈ ran (coursename) THEN
resp := invalid_course
ELSE ANY aid, course WHERE aid = assessorname^{-1} (asname) \land
course = coursename^{-1} (csname) THEN
SELECT course /∈ assesses [\{aid\}] THEN
resp := not_authorized
WHEN sid /∈ students THEN
resp := invalid_student
WHEN course /∈ courseenrolled [\{sid\}] THEN
resp := not_enrolled_course
ELSE resp := OK \parallel completeCourse (aid, sid, course)
END
END
END
END;
resp ← Failcourse (asname, sid, csname) ∨
PRE asname ∈ ASSESSORNAME \land sid ∈ STUDENT \land
cid ∈ COURSENAME THEN
SELECT asname /∈ ran (assessorname) THEN
resp := invalid_assessor
WHEN csname /∈ ran (coursename) THEN
resp := invalid_course
ELSE ANY aid, course WHERE aid = assessorname^{-1} (asname) \land
course = coursename^{-1} (csname) THEN
SELECT sid /∈ students THEN
resp := invalid_student
WHEN course /∈ courseenrolled [\{sid\}] THEN
resp := not_enrolled_course
ELSE resp := OK \parallel failcourse (aid, sid, course)
END
END
END
END;
resp ← CompleteProg (sid) ∨
PRE sid ∈ STUDENT THEN
SELECT sid /∈ dom (progenrolled) THEN
resp := not_enrolled_student

WHEN sid ∈ dom ( courseenrolled ) THEN
  resp := courses_enrolled
WHEN ¬( completed ( sid , progenrolled ( sid ) ) ) THEN
  resp := program_not_completed
ELSE  resp := OK || completeProg ( sid )
END

END ;
resp ← SaveProgramManager ≜
BEGIN  resp ∈ { OK , save_failed } || skip END ;

resp ← RestoreProgramManager ≜
VAR  bb IN
  resp ∈ { OK , restore_failed } ||
  bb ← restoreassessor
END

END

B Implementation

B.1 The Implementation of ProgramManagerAPI

IMPLEMENTATION ProgramManagerAPII
REFINES ProgramManagerAPI
SEES Bool_TYPE , String_TYPE
IMPORTS ProgramManager ( maxAssessor , maxStudent , maxProg ,
  maxCourse , maxPrereqs ) ,
AssessorName , Assessor_CTX ,
Student_CTX ,
ProgName , Program_CTX ,
CourseName , Course_CTX

PROPERTIES
  COURSENAME = STRING ∧
  PROGNAME = STRING ∧
  ASSESSORNAME = STRING

OPERATIONS
  resp ← NewCourse ( csname ) ≜
  VAR  bb IN
    bb ← newCourseName ( csname ) ;
    resp := existing_course_name ;
    IF  bb = TRUE THEN
      bb ← MoreCourses ;
      resp := course_limit_reached ;
    IF  bb = TRUE THEN
      resp := OK ; newCourse ( csname )
    END
  END

END
END;
resp \leftarrow \text{NewgeCourse}(csname) \\
\text{VAR} \ bb,\ course \ \text{IN} \\
bb,\ course \leftarrow \text{CourseExists}(csname) ; \\
resp := \text{no\_such\_course} ; \\
\text{IF} \ bb = \text{TRUE} \ \text{THEN} \\
bb \leftarrow \text{isGenEd}(course) ; \\
resp := \text{already\_gened} ; \\
\text{IF} \ bb = \text{FALSE} \ \text{THEN} \\
\begin{align*}
resp & := \text{OK} ; \text{newgeCourse}(course) \\
\end{align*}
\text{END}
\text{END}
\text{END} ;
resp \leftarrow \text{Delgecourse}(csname) \\
\text{VAR} \ bb,\ course \ \text{IN} \\
bb,\ course \leftarrow \text{CourseExists}(csname) ; \\
resp := \text{no\_such\_course} ; \\
\text{IF} \ bb = \text{TRUE} \ \text{THEN} \\
bb \leftarrow \text{isGenEd}(course) ; \\
resp := \text{not\_gened} ; \\
\text{IF} \ bb = \text{TRUE} \ \text{THEN} \\
\begin{align*}
resp & := \text{OK} ; \text{delgecourse}(course) \\
\end{align*}
\text{END}
\text{END}
\text{END} ;
resp \leftarrow \text{AddPrereq}(csname,\ prename) \\
\text{VAR} \ bb,\ course,\ prereq \ \text{IN} \\
bb,\ course \leftarrow \text{CourseExists}(csname) ; \\
resp := \text{no\_such\_course} ; \\
\text{IF} \ bb = \text{TRUE} \ \text{THEN} \\
bb,\ prereq \leftarrow \text{CourseExists}(prename) ; \\
resp := \text{non\_existing\_prereq} ; \\
\text{IF} \ bb = \text{TRUE} \ \text{THEN} \\
bb \leftarrow \text{InPrereqs}(course,\ prereq) ; \\
resp := \text{already\_prereq} ; \\
\text{IF} \ bb = \text{FALSE} \ \text{THEN} \\
\text{IF} \ course = \text{prereq} \ \text{THEN} \\
resp := \text{reflexive\_prereq} \\
\text{ELSE} \ bb \leftarrow \text{ACyclicPrereq}(course,\ prereq) ; \\
resp := \text{cyclic\_prereq} ; \\
\text{IF} \ bb = \text{TRUE} \ \text{THEN} \\
\begin{align*}
resp & := \text{OK} ; \text{addPrereq}(course,\ prereq) \\
\end{align*}
\text{END}
\text{END}
\text{END}
\text{END}
resps ← DelPrereq (csname, prename) ≜
VAR bb, course, prereq IN
bb, course ← CourseExists (csname);
resp := no_such_course;
IF bb = TRUE THEN
bb, prereq ← CourseExists (prename);
resp := non_existing_prereq;
IF bb = TRUE THEN
bb ← InPrereqs (course, prereq);
resp := not_a_prereq;
IF bb = TRUE THEN
resp := OK; delPrereq (course, prereq)
END
END
END

resps ← FirstPreReq (csname) ≜
VAR bb, course, pre IN
bb, course ← CourseExists (csname);
resp := no prereqs; prename ← anyCOURSENAME;
IF bb = TRUE THEN
bb, pre ← firstPreReq (course);
resp := no prereqs;
IF bb = TRUE THEN
resp := OK; prename ← courseName (pre)
END
END
END

resps ← NextPreReq (csname) ≜
VAR bb, course, pre IN
bb, course ← CourseExists (csname);
resp := no prereqs; prename ← anyCOURSENAME;
IF bb = TRUE THEN
bb, pre ← nextPreReq (course);
resp := no more prereqs;
IF bb = TRUE THEN
resp := OK; prename ← courseName (pre)
END
END
END

resps ← NewProgram (prname) ≜
VAR bb IN
bb ← newProgName (prname);
resp := existing_program_name;
IF bb = TRUE THEN
bb ← MorePrograms;

37
resp := program_limit_reached;

IF bb = TRUE THEN
    resp := OK; newProgram ( prname )
END

END

resp ← AddCore ( prname , csname )

VAR bb , program , course IN
bb , program ← ProgramExists ( prname ) ;
resp := invalid_program ;
IF bb = TRUE THEN
    bb , course ← CourseExists ( csname ) ;
    resp := invalid_course ;
    IF bb = TRUE THEN
        bb ← isGenEd ( course ) ;
        resp := course_in_gened ;
        IF bb = FALSE THEN
            bb ← isElective ( program , course ) ;
            resp := course_in_elective ;
            IF bb = FALSE THEN
                resp := OK ; addCore ( program , course )
            END
        END
    END
END

END

resp ← DelCore ( prname , csname )

VAR bb , course , program IN
bb , program ← ProgramExists ( prname ) ;
resp := invalid_program ;
IF bb = TRUE THEN
    bb , course ← CourseExists ( csname ) ;
    resp := invalid_course ;
    IF bb = TRUE THEN
        bb ← isCore ( program , course ) ;
        resp := course_not_core ;
        IF bb = TRUE THEN
            resp := OK ; delCore ( program , course )
        END
    END
END

END

resp ← AddElective ( prname , csname )

VAR bb , course , program IN
bb , program ← ProgramExists ( prname ) ;
resp := invalid_program ;
IF bb = TRUE THEN
bb, course ← CourseExists (csname);
resp := invalid_course;

IF bb = TRUE THEN
bb ← isGenEd (course);
resp := course_in_gened;

IF bb = FALSE THEN
bb ← isCore (program, course);
resp := course_in_core;

IF bb = FALSE THEN
resp := OK; addElective (program, course)
END
END
END

VAR bb, course, program IN

bb, course ← CourseExists (csname);
resp := invalid_course;

IF bb = TRUE THEN
bb ← isGenEd (course);
resp := course_in_gened;

IF bb = TRUE THEN
bb ← isCore (program, course);
resp := course_in_core;

IF bb = TRUE THEN
resp := OK; addElective (program, course)
END
END
END

resp ← DelElective (prname, csname) ≥

VAR bb, course, program IN

bb, program ← ProgramExists (prname);
resp := invalid_program;

IF bb = TRUE THEN
bb, course ← CourseExists (csname);
resp := invalid_course;

IF bb = TRUE THEN
bb ← isElective (program, course);
resp := course_not_elective;

IF bb = TRUE THEN
resp := OK; delElective (program, course)
END
END
END

resp ← SetElectCount (prname, count) ≥

VAR bb, program IN

bb, program ← ProgramExists (prname);
resp := invalid_program;

IF bb = TRUE THEN
resp := OK; setElectCount (program, count)
END

resp ← SetGECount (prname, count) ≥

VAR bb, program IN

bb, program ← ProgramExists (prname);
resp := invalid_program;

IF bb = TRUE THEN
resp := OK; setGECount (program, count)
END

END
END

39
resp, sid ← NewStudent ≡

VAR bb IN
bb ← moreStudents;
resp := student_limit_reached; sid ← anyStudent;
IF bb = TRUE THEN
  resp := OK; sid ← newStudent
END
END;
resp ← EnrolProg ( sid, prname ) ≡
VAR bb, program IN
bb, program ← ProgramExists ( prname );
resp := invalid_program;
IF bb = TRUE THEN
  bb ← studentEnrolled ( sid );
  resp := already_enrolled;
IF bb = FALSE THEN
  bb ← validStudent ( sid );
  resp := invalid_student;
IF bb = TRUE THEN
  resp := OK; enrolProg ( sid, program )
END
END
END;
resp ← EnrolCourse ( sid, csnme ) ≡
VAR bb, course IN
bb, course ← CourseExists ( csnme );
resp := invalid_course;
IF bb = TRUE THEN
  bb ← studentEnrolled ( sid );
  resp := not_enrolled_student;
IF bb = TRUE THEN
  bb ← courseEnrolled ( sid, course );
  resp := already_enrolled;
IF bb = FALSE THEN
  bb ← courseCompleted ( sid, course );
  resp := course_completed;
IF bb = FALSE THEN
  bb ← prereqsCompleted ( sid, course );
  resp := missing_prereqs;
IF bb = TRUE THEN
  bb ← inProgram ( sid, course );
  resp := course_not_in_program;
IF bb = TRUE THEN
  bb ← coreCompleted ( sid );
  resp := core_completed;
IF bb = FALSE THEN

bb ← electiveCompleted (sid);
resp := electives_completed;

IF bb = FALSE THEN
    bb ← genedCompleted (sid);
    resp := gened_completed;
    IF bb = FALSE THEN
        resp := OK; enrolCourse (sid, course)
    END
END
END
END
END

resp ← DisenrolCourse (sid, csname) ≝
VAR bb, course IN
    bb ← studentEnrolled (sid);
    resp := not_enrolled_student;
    IF bb = TRUE THEN
        resp := not_enrolled_course;
        bb, course ← CourseExists (csname);
        IF bb = TRUE THEN
            bb ← courseEnrolled (sid, course);
            IF bb = TRUE THEN
                resp := OK; disenrolCourse (sid, course)
            END
        END
    END
    END
END
END

resp ← NewAssessor (asname) ≝
VAR bb IN
    bb ← newAssessorName (asname);
    resp := existing_assessor_name;
    IF bb = TRUE THEN
        bb ← MoreAssessors;
        resp := assessor_limit_reached;
        IF bb = TRUE THEN
            resp := OK; newAssessor (asname)
        END
    END
END
END

resp ← AddAssessor (asname, csname) ≝
VAR bb, course, aid IN
    bb, aid ← validAssessor (asname);
resp := invalid_assessor;

IF bb = TRUE THEN
    bb, course ← CourseExists (csname);
    resp := invalid_course;
    IF bb = TRUE THEN
        resp := OK; addAssessor (aid, course)
    END
END
END

resp ← DelAssessor (asname, csname) ≡
VAR bb, course, aid IN
    bb, aid ← validAssessor (asname);
    resp := invalid_assessor;
    IF bb = TRUE THEN
        bb, course ← CourseExists (csname);
        resp := not_authorised;
        IF bb = TRUE THEN
            bb ← AssessorAccredited (aid, course);
            IF bb = TRUE THEN
                resp := OK; delAssessor (aid, course)
            END
        END
    END
    END
END
END

resp ← CompleteCourse (asname, sid, csname) ≡
VAR bb, aid, course IN
    bb, aid ← validAssessor (asname);
    resp := invalid_assessor;
    IF bb = TRUE THEN
        bb, course ← CourseExists (csname);
        resp := not_authorised;
        IF bb = TRUE THEN
            bb ← validStudent (sid);
            IF bb = TRUE THEN
                resp := invalid_student;
                IF bb = TRUE THEN
                    bb ← courseEnrolled (sid, course);
                    resp := not_enrolled_course;
                    IF bb = TRUE THEN
                        resp := OK; completeCourse (aid, sid, course)
                    END
                END
            END
        END
    END
END
END
END
END
END
resp ← Failcourse ( asname , sid , csname )
VAR bb , aid , course IN
bb , aid ← validAssessor ( asname ) ;
resp := invalid_assessor ;
IF bb = TRUE THEN
bb , course ← CourseExists ( csname ) ;
resp := not_authorised ;
IF bb = TRUE THEN
bb ← AssessorAccredited ( aid , course ) ;
IF bb = TRUE THEN
bb ← validStudent ( sid ) ;
resp := invalid_student ;
IF bb = TRUE THEN
bb ← courseEnrolled ( sid , course ) ;
resp := not_enrolled_course ;
IF bb = TRUE THEN
resp := OK ; failcourse ( aid , sid , course )
END
END
END
END
END ;
resp ← CompleteProg ( sid )
VAR bb IN
bb ← studentEnrolled ( sid ) ;
resp := not_enrolled_student ;
IF bb = TRUE THEN
bb ← coursesEnrolled ( sid ) ;
resp := courses_enrolled ;
IF bb = FALSE THEN
bb ← ProgramCompleted ( sid ) ;
resp := program_not_completed ;
IF bb = TRUE THEN
resp := OK ; completeProg ( sid )
END
END
END
END ;
resp ← SaveProgramManager
VAR bb IN
bb ← saveassessor ;
IF bb = TRUE THEN
resp := OK
ELSE resp := save_failed
END
END ;
resp ← RestoreProgramManager ≜
  VAR bb IN
  bb ← restoreassessor ;
  IF bb = TRUE THEN
    resp := OK
  ELSE resp := restore_failed
  END
END

B.2 Implementation of GenEd/Course

The base specification, CourseDB.bse is used by the BToolkit to generate a base machine, CourseDB, a context machine, CourseDBCtx, and their implementations. To implement GenEd we import CourseDB and see CourseDBCtx. Note that CourseDB will have parameters that must be instantiated. The available operations for CourseDB can be found in CourseDB.ops in the Generator environment. There is chicken and egg problem initially, because until the operations are used in the GenEdI machine they will not be selected for the CourseDB machine. Remake should fix this.

SYSTEM CourseDB
SUPPORTS GenEdI
IS
  GLOBAL
    GenEd ∈ set ( COURSEDB ) [ 100 ]
END ;

BASE COURSEDB
MANDATORY
  CourseName ∈ STRING [ 50 ] ;
  PreReqs ∈ SEQ ( COURSEDB ) [ 100 ] ;
  PreReqClosure ∈ SET ( COURSEDB ) [ 100 ] ;
  remPreReqs ∈ ℕ
END

IMPLEMENTATION GenEdI
REFINES GenEd
SEES
  CourseName, Course_CTX ,
  CourseDBCtx ,
  Bool_TYPE , Bool_TYPE_Ops ,
  Scalar_TYPE , String_TYPE
IMPORTS CourseDB ( maxCourse )
PROPERTIES
  COURSEDB_ABSOBJ = COURSE ∧
  COURSENAME = STRING
INvariant
  CourseName ∈ COURSEDB ⇒ COURSENAME ∧
\[ \text{COURSEDB} = \text{courses} \land \\
\text{GenEd} = \text{gened} \land \\
\text{CourseName} = \text{coursename} \land \\
\forall \text{cs . (cs} \in \text{COURSEDB} \Rightarrow \\
\text{ran (PreReqs (cs))} = \text{prereqs [ { cs } ]} \land \\
\forall \text{cs . (cs} \in \text{COURSEDB} \Rightarrow \\
\text{PreReqs (cs)[1 . . . \text{remPreReqs (cs)}]} = \text{rempreqs [ { cs } ]} \land \\
\forall \text{cs . (cs} \in \text{COURSEDB} \Rightarrow \\
\text{PreReqClosure (cs)} = \text{prereqclosure [ { cs } ]} \)
\]

\text{OPERATIONS}

\text{newgeCourse (course) } \equiv \\
\begin{align*}
\text{VAR } & \text{bb IN} \\
& \text{bb} \leftarrow \text{add_GenEd (course)} \\
\end{align*}
\text{END ;}

\text{dellgecourse (course) } \equiv \\
\begin{align*}
& \text{del_GenEd (course)} \\
& \text{ok } \leftarrow \text{isGenEd (course)} \\
& \text{ok } \leftarrow \text{member_GenEd (course)} ;
\end{align*}

\text{newCourse (coursename) } \equiv \\
\begin{align*}
\text{VAR } & \text{bb, db IN} \\
& \text{bb, db} \leftarrow \text{make_COURSEDB (0)} ; \\
& \text{bb} \leftarrow \text{mod_CourseName (db, coursename)} \\
\end{align*}
\text{END ;}

\text{addPrereq (course, prereq) } \equiv \\
\begin{align*}
\text{VAR } & \text{bb IN} \\
& \text{bb} \leftarrow \text{push_PreReqs (course, prereq)} ; \\
& \text{mod_rempreqs (course, 0)} \\
\end{align*}
\text{END ;}

\text{delPrereq (course, prereq) } \equiv \\
\begin{align*}
\text{VAR } & \text{bb, ii IN} \\
& \text{bb, ii} \leftarrow \text{within_PreReqs (course, prereq)} ; \\
& \text{bb} \leftarrow \text{delIth_PreReqs (course, ii)} \\
\end{align*}
\text{END ;}

\text{ok, prereq } \leftarrow \text{firstPreReq (course) } \equiv \\
\begin{align*}
\text{VAR } & \text{nn IN} \\
& \text{nn} \leftarrow \text{length_PreReqs (course)} ; \\
& \text{IF } \text{nn} \neq 0 \text{ THEN} \\
& \text{ok } := \text{TRUE} ; \\
& \text{prereq} \leftarrow \text{last_PreReqs (course)} ; \\
& \text{mod_rempreqs (course, nn }-1 \text{) ELSE} \\
& \text{ok } := \text{FALSE} ; \text{prereq } \leftarrow \text{any_COURSEDB} \\
\end{align*}
\text{END}
\text{END ;}

\text{ok, prereq } \leftarrow \text{nextPreReq (course) } \equiv \\
\begin{align*}
\text{VAR } & \text{nn IN} \\
& \text{nn} \leftarrow \text{val_rempreqs (course)} ;
\end{align*}
IF \( nn \neq 0 \) THEN
\[
\text{ok} := \text{TRUE} ;
\]
prereq \( \leftarrow \text{valth\_PreReqs (course, nn)} ;
\]
\[
\text{mod\_remPreReqs (course, nn - 1)}
\]
ELSE
\[
\text{ok} := \text{FALSE} ;
\]
\[
\text{prereq} \leftarrow \text{any\_COURSEDB}
\]
END

END;

casename \( \leftarrow \text{courseName (course)} \) \( \triangleq \)
casename \( \leftarrow \text{xir\_CourseName (course)} ;
\]
ok \( \leftarrow \text{MoreCourses} \equiv \)
VAR nbr IN
\[
\text{nbr} \leftarrow \text{nbr\_COURSEDB} ;
\]
\[
\text{IF} \quad \text{nbr} < \text{maxCourse}
\]
THEN
\[
\text{ok} := \text{TRUE}
\]
ELSE
\[
\text{ok} := \text{FALSE}
\]
END
END;
ok, csid \( \leftarrow \text{CourseExists (casname)} \equiv \)
ok, csid \( \leftarrow \text{key\_search\_CourseName (csname)} ;
\]
ok \( \leftarrow \text{newCourseName (name)} \equiv \)
VAR bb, db IN
\[
\text{bb} , \text{db} \leftarrow \text{key\_search\_CourseName (name)} ;
\]
ok \( \leftarrow \text{NEG\_BOOL (bb)} \)
END;
ok \( \leftarrow \text{InPrereqs (course, prereq)} \equiv \)
VAR ii IN
\[
\text{ok} , \text{ii} \leftarrow \text{within\_PreReqs (course, prereq)}
\]
END;
ok \( \leftarrow \text{ACyclicPrereq (course, prereq)} \equiv \)
VAR bb IN
\[
\text{bb} \leftarrow \text{member\_PreReqClosure (course, prereq)} ;
\]
ok \( \leftarrow \text{NEG\_BOOL (bb)} \)
END;
ok \( \leftarrow \text{HasPrereqs (course)} \equiv \)
VAR nbr IN
\[
\text{nbr} \leftarrow \text{length\_PreReqs (course)} ;
\]
\[
\text{IF} \quad \text{nbr} \neq 0
\]
THEN
\[
\text{ok} := \text{TRUE}
\]
ELSE
\[
\text{ok} := \text{FALSE}
\]
END
END;
ok \( \leftarrow \text{MorePrereqs (course)} \equiv \)
VAR nn IN
\[
\text{nn} \leftarrow \text{val\_remPreReqs (course)} ;
\]
\[
\text{IF} \quad \text{nn} \neq 0
\]
THEN
\[
\text{ok} := \text{TRUE}
\]
ELSE
\[
\text{ok} := \text{FALSE}
\]
B.3 Implementation of Program

SYSTEM ProgramDB
SUPPORTS ProgramI
IS

BASE PROGRAMDB
MANDATORY

ProgName ∈ STRING [ 100 ] ;
ProgCore ∈ SEQ ( COURSE ) [ 100 ] ;
Elective ∈ SEQ ( COURSE ) [ 100 ] ;
ElectCount ∈ N ;
GECount ∈ N ;
remProgCore ∈ N ;
remElective ∈ N ;

END

IMPLEMENTATION ProgramI
REFINES Program
SEES

ProgName , Program_CTX ,
CourseName , Course_CTX ,
Bool_TYPE , Bool_TYPE_Ops ,
Scalar_TYPE , String_TYPE ,
ProgramDBCtx
IMPORTS
ProgramDB ( maxProg , COURSE ) ,
GenEd ( maxCourse , maxPrereqs )
PROMOTES
newCourse , addPrereq , delPrereq , firstPreReq ,
nextPreReq , MoreCourses , anyCourse , newCourseName ,
CourseExists , courseName , InPrereqs , ACyclicPrereq ,
HasPrereqs , MorePrereqs , newgeCourse , delgecourse ,
isGenEd , savegened , restoregened
PROPERTIES
PROGRAM = PROGRAMDB_ABSOBJ ∧
COURSENAME = STRING ∧
PROGNAME = STRING
INVARIANT

\( \text{PROGRAMDB} = \text{programs} \land \)
\( \text{ProgName} = \text{progname} \land \)
\( \forall \text{pr} . (\text{pr} \in \text{PROGRAMDB}) \Rightarrow \)
\( \text{ran} (\text{ProgCore} (\text{pr})) = \text{progcore} [\{ \text{pr} \}] \land \)
\( \forall \text{pr} . (\text{pr} \in \text{PROGRAMDB}) \Rightarrow \)
\( \text{ProgCore} (\text{pr}) [1 \ldots \text{remProgCore} (\text{pr})] = \text{remprogcore} [\{ \text{pr} \}] \land \)
\( \forall \text{pr} . (\text{pr} \in \text{PROGRAMDB}) \Rightarrow \)
\( \text{ran} (\text{Elective} (\text{pr})) = \text{elective} [\{ \text{pr} \}] \land \)
\( \forall \text{pr} . (\text{pr} \in \text{PROGRAMDB}) \Rightarrow \)
\( \text{Elective} (\text{pr}) [1 \ldots \text{remElective} (\text{pr})] = \text{remelective} [\{ \text{pr} \}] \land \)
\( \text{ElectCount} = \text{electcount} \land \)
\( \text{GECount} = \text{gecount} \)

OPERATIONS

\text{newProgram} (\text{prname}) \equiv
\begin{align*}
\text{VAR} & \quad \text{bb, db IN} \\
& \quad \text{bb, db} \leftarrow \text{makePROGRAMDB} (0, 0, 0, 0) ; \\
& \quad \text{bb} \leftarrow \text{modProgName} (\text{db, prname}) \\
\end{align*}
\text{END} ;

\text{addCore} (\text{program, course}) \equiv
\begin{align*}
\text{VAR} & \quad \text{bb IN} \\
& \quad \text{bb} \leftarrow \text{pushProgCore} (\text{program, course}) ; \\
& \quad \text{mod_remProgCore} (\text{program, 0}) \\
\end{align*}
\text{END} ;

\text{delCore} (\text{program, course}) \equiv
\begin{align*}
\text{VAR} & \quad \text{bb, ii IN} \\
& \quad \text{bb, ii} \leftarrow \text{withinProgCore} (\text{program, course}) ; \\
& \quad \text{bb} \leftarrow \text{delIthProgCore} (\text{program, ii}) \\
\end{align*}
\text{END} ;

\text{addElective} (\text{program, course}) \equiv
\begin{align*}
\text{VAR} & \quad \text{bb IN} \\
& \quad \text{bb} \leftarrow \text{pushElective} (\text{program, course}) ; \\
& \quad \text{mod_remElective} (\text{program, 0}) \\
\end{align*}
\text{END} ;

\text{delElective} (\text{program, course}) \equiv
\begin{align*}
\text{VAR} & \quad \text{bb, ii IN} \\
& \quad \text{bb, ii} \leftarrow \text{withinElective} (\text{program, course}) ; \\
& \quad \text{bb} \leftarrow \text{delIthElective} (\text{program, ii}) \\
\end{align*}
\text{END} ;

\text{setElectCount} (\text{program, count}) \equiv
\begin{align*}
& \text{mod_ElectCount} (\text{program, count}) ; \\
\end{align*}
\text{setGECount} (\text{program, count}) \equiv
\begin{align*}
& \text{mod_GECount} (\text{program, count}) ; \\
\end{align*}
\text{prname} \leftarrow \text{progName} (\text{program}) \equiv
\begin{align*}
& \text{prname} \leftarrow \text{xrProgName} (\text{program}) ; \\
\end{align*}
\text{ok} \leftarrow \text{MorePrograms} \equiv
\begin{align*}
& \text{VAR} \quad \text{nbr IN} \\
\end{align*}
nbr ← nbr_PROGRAMDB ;
IF nbr ≠ maxProg THEN
  ok := TRUE
ELSE ok := FALSE
END
END ;
ok ← newProgName ( prname ) =
VAR bb, pr IN
  bb, pr ← key_search_ProgName ( prname ) ;
  ok ← NEG_BOOL ( bb )
END ;
ok, program ← ProgramExists ( prname ) =
ok, program ← key_search_ProgName ( prname ) ;
ok ← isCore ( program , course ) =
VAR ii IN
  ok, ii ← within_ProgCore ( program , course )
END ;
ok ← isElective ( program , course ) =
VAR ii IN
  ok, ii ← within_Elective ( program , course )
END ;
ok, course ← firstCore ( program ) =
VAR nn IN
  nn ← length_ProgCore ( program ) ;
IF nn ≠ 0 THEN
  ok := FALSE ;
  course ← last_ProgCore ( program ) ;
  mod_remProgCore ( program , nn − 1 )
ELSE
  ok := TRUE ;
  course ← anyCourse
END
END ;
ok, course ← nextCore ( program ) =
VAR nn IN
  nn ← val_remProgCore ( program ) ;
  course ← valIth_ProgCore ( program , nn ) ;
IF nn ≠ 0 THEN
  ok := TRUE ;
  mod_remProgCore ( program , nn − 1 )
ELSE ok := FALSE
END
END ;
ok, course ← firstElective ( program ) =
VAR nn IN
  nn ← length_Elective ( program ) ;
IF nn ≠ 0 THEN
  ok := TRUE ;
course ← last_Electic ( program ) ;
mod_remElective ( program , nn − 1 )
ELSE
  ok := FALSE ; course ← anyCourse
END
END ;
ok , course ← nextElective ( program ) =
VAR nn IN
  nn ← val_remElective ( program ) ;
course ← valIth_Electic ( program , nn ) ;
IF nn ≠ 0 THEN
  ok := TRUE ;
mod_remElective ( program , nn − 1 )
ELSE ok := FALSE
END
END

count ← progElectCount ( program ) =
count ← val_ElectCount ( program ) ;
count ← progGECount ( program ) =
count ← val_GECount ( program ) ;
ok ← saveprogram =
VAR bb , cc IN
  bb ← savegened ;
  cc ← save_ProgramDB ;
  ok ← CNJ_BOOL ( bb , cc )
END ;
ok ← restoreprogram =
VAR bb , cc IN
  bb ← restoregened ;
  cc ← restore_ProgramDB ;
  ok ← CNJ_BOOL ( bb , cc )
END

B.4 Implementation of Student

SYSTEM StudentDB
SUPPORTS StudentI
IS

BASE STUDENTDB
MANDATORY
  ProgCompleted ∈ SET ( PROGRAM ) [ 100 ] ;
  CourseEnrolled ∈ SET ( COURSE ) [ 100 ] ;
  CourseCompleted ∈ SET ( COURSE ) [ 100 ] ;
  ElectiveCompleted ∈ N ;
  GenedCompleted ∈ N ;
  CourseFailed ∈ SET ( COURSE ) [ 100 ] ;
OPTIONAL

ProgEnrolled ∈ PROGRAM

END

IMPLEMENTATION  StudentI

REFINES  Student

SEE

Student_CTX, 
ProgName , Program_CTX, 
CourseName , Course_CTX, 
Bool_TYPE , Bool_TYPE_Ops, 
Scalar_TYPE , String_TYPE,

StudentDBCtx

IMPORTS

StudentDB ( maxStudent , PROGRAM , COURSE ), 
Program ( maxProg , maxCourse , maxPrereqs )

PROMOTES

newCourse , addPrereq , delPrereq , firstPreReq, 
nextPreReq , MoreCourses , anyCourse , newCourseName , 
CourseExists , courseName , InPrereqs , ACyclicPrereq, 
HasPrereqs , MorePrereqs , newgeCourse , delgecourse, 
isGenEd , savegeened , restoregeened , newProgram , addCore, 
delCore , addElective , delElective , setElectCount, 
setGECount , progName , MorePrograms , newProgName, 
ProgramExists , isCore , isElective , firstCore, 
nextCore , firstElective , nextElective , progElectCount , 
progGECount , saveprogram , restoreprogram

PROPERTIES

STUDENT = STUDENTDB абсOBJ ∧

COURSENAME = STRING ∧

PROGNAME = STRING

INVARIANT

STUDENTDB = students ∧

ProgEnrolled = progenrolled ∧

∀ st . ( st ∈ STUDENTDB ⇒

ProgCompleted ( st ) = progcompleted [ { st } ] ) ∧

∀ st . ( st ∈ STUDENTDB ⇒

CourseEnrolled ( st ) = courseenrolled [ { st } ] ) ∧

∀ st . ( st ∈ STUDENTDB ⇒

CourseCompleted ( st ) = coursecompleted [ { st } ] ) ∧

∀ st . ( st ∈ STUDENTDB ⇒

CourseFailed ( st ) = coursefailed [ { st } ] ) ∧

ElectiveCompleted = electivecompleted ∧

GenedCompleted = genedcompleted

OPERATIONS

student ← newStudent

VAR  bb IN
debug, student ← make_STUDENTDB (0, 0)

END;
enrolProg (student, program) ≡
BEGIN
cre_ProgEnrolled (student, program);
mod_ElectiveCompleted (student, 0);
mod_GenedCompleted (student, 0)
END;
enrolCourse (student, course) ≡
VAR bb IN
    bb ← add_CourseEnrolled (student, course)
END;
disenrolCourse (student, course) ≡
del_CourseEnrolled (student, course);
completeCourse0 (student, course) ≡
VAR bb, pr, nn IN
    bb ← add_CourseCompleted (student, course);
    bb ← isGenEd (course);
    IF bb = TRUE THEN
        nn ← val_GenedCompleted (student);
        mod_GenedCompleted (student, nn + 1)
    ELSE
        pr ← val_ProgEnrolled (student);
        bb ← isElective (pr, course);
        IF bb = TRUE THEN
            nn ← val_ElectiveCompleted (student);
            mod_ElectiveCompleted (student, nn + 1)
    END
END
END;
failCourse0 (student, course) ≡
VAR bb IN
    bb ← add_CourseFailed (student, course)
END;
completeProg0 (student, program) ≡
VAR bb IN
    bb ← add_ProgCompleted (student, program);
    rem_ProgEnrolled (student);
    mod_ElectiveCompleted (student, 0);
    mod_GenedCompleted (student, 0)
END;
ok ← moreStudents ≡
VAR nbr IN
    nbr ← nbr_STUDENTDB;
    IF nbr ≠ maxStudent THEN
        ok := TRUE
    ELSE ok := FALSE
END

52
END;
ok ← validStudent (student) ≡
ok ← vld_STUDENTDB (student) ;
ok ← studentEnrolled (student) ≡
ok ← def_ProgEnrolled (student) ;
ok ← inProgram (student, course) ≡

VAR bb , pr IN
pr ← val_ProgEnrolled (student) ;
bb ← isCore (pr , course) ;
IF bb = TRUE THEN
ok := TRUE
ELSE
bb ← isElective (pr , course) ;
IF bb = TRUE THEN
ok := TRUE
ELSE
ok ← isGenEd (course)
END
END
END;
ok ← courseEnrolled (student, course) ≡
ok ← member_CourseEnrolled (student, course) ;
ok ← courseCompleted (student, course) ≡
ok ← prereqsCompleted (student, course) ≡

VAR bb , pr , cp IN
bb , pr ← firstPreReq (course) ; cp := TRUE ;
WHILE bb = TRUE ∧ cp = TRUE DO
  cp ← member_CourseCompleted (student, pr ) ;
  bb , pr ← nextPreReq (course)
INVARIANT
( bb = TRUE ⇒ pr ∈ prereqs [ { course } ] ) ∧
( bb = TRUE ∧ cp = TRUE ⇒
  prereqs [ { course } ] − remprereqs [ { course } ] ∪ { pr } ⊆
  coursecompleted [ { student } ] ) ∧
( bb = FALSE ∧ cp = TRUE ⇒
  prereqs [ { course } ] ⊆ coursecompleted [ { student } ] ) ∧
( bb = FALSE ⇒ remprereqs [ { course } ] = {} ) ∧
( cp = FALSE ⇒ pr ∉ coursecompleted [ { student } ] )
VARIANT card (remprereqs [ { course } ])
END;
ok := cp
END;
ok ← coreCompleted (student) ≡

VAR bb , pr , cr , cp IN
pr ← val_ProgEnrolled (student) ;
bb , cr ← firstCore (pr ) ; cp := TRUE ;
WHILE bb = TRUE ∧ cp = TRUE DO
cp ← member_CourseCompleted ( student, cr ) ;  
bb, cr ← nextCore ( pr )

ININVARIANT

pr = progEnrolled ( student ) ∧ 
( bb = TRUE ⇒ cr ∈ progcore [ pr ] ) ∧ 
( bb = TRUE ∧ cp = TRUE ⇒ 
  progcore [ pr ] − remprogcore [ pr ] ∪ { cr } ⊆ 
  coursecompleted [ student ] ) ∧ 
( bb = FALSE ∧ cp = TRUE ⇒ 
  progcore [ pr ] ⊆ coursecompleted [ student ] ) ∧ 
( bb = FALSE ⇒ remprogcore [ pr ] = {} ) ∧ 
( cp = FALSE ⇒ cr ∉ coursecompleted [ student ] )

VARIANT card ( remprogcore [ pr ] )

END ;

ok := cp

END ;

ok ← electiveCompleted ( student ) ≜ 
VAR pr, req, comp IN 
   pr ← val_ProgEnrolled ( student ) ; 
   req ← progElectCount ( pr ) ; 
   comp ← val_ElectiveCompleted ( student ) ; 
   ok := FALSE ;  
   IF req ≤ comp THEN 
     ok := TRUE
   END

END ;

ok ← genedCompleted ( student ) ≜ 
VAR pr, req, comp IN 
   pr ← val_ProgEnrolled ( student ) ; 
   req ← progGECount ( pr ) ; 
   comp ← val_GenedCompleted ( student ) ; 
   ok := FALSE ;  
   IF req ≤ comp THEN 
     ok := TRUE
   END

END ;

ok ← coursesEnrolled ( student ) ≜ 
VAR nbr IN 
   nbr ← card_CourseEnrolled ( student ) ; 
   ok := TRUE ;  
   IF nbr = 0 THEN 
     ok := FALSE
   END

END ;

program ← programEnrolled ( student ) ≜ 
VAR pr IN 
   pr ← val_ProgEnrolled ( student )

54
END;

student ← anyStudent ≡
BEGIN
    student ← any_STUDENTDB
END;

ok ← savestudent ≡
VAR bb, cc IN
    bb ← savegened ;
    cc ← save_StudentDB ;
    ok ← CNJ_BOOL ( bb , cc )
END;

ok ← restorestudent ≡
VAR bb, cc IN
    bb ← restoreprogram ;
    cc ← restore_StudentDB ;
    ok ← CNJ_BOOL ( bb , cc )
END

B.5 Implementation of Assessor

SYSTEM AssessorDB
SUPPORTS AssessorI
IS

BASE ASSESSORDB
MANDATORY
    AssessorName ∈ STRING [ 100 ] ;
    Assesses ∈ SET ( COURSE ) [ 100 ]

END
END

IMPLEMENTATION AssessorI
REFINES Assessor
SEES
    AssessorName , Assessor_CTX ,
    Student_CTX ,
    ProgName , Program_CTX ,
    CourseName , Course_CTX ,
    Bool_TYPE , Bool_TYPE_Ops ,
    Scalar_TYPE , String_TYPE ,
    AssessorDBCtx
IMPORTS
    AssessorDB ( maxAssessor , COURSE ) ,
    Student ( maxStudent , maxProg , maxCourse , maxPrereqs )
PROMOTES
    newCourse , addPrereq , delPrereq , firstPreReq ,
    nextPreReq , MoreCourses , anyCourse , newCourseName ,

CourseExists, courseName, InPrereqs, ACyclicPrereq, HasPrereqs, MorePrereqs, newgeCourse, delgecourse, isGenEd, savegened, restoregened, newProgram, addCore, delCore, addElective, delElective, setElectCount, setGECount, progName, MorePrograms, newProgName, ProgramExists, isCore, isElective, firstCore, nextCore, firstElective, nextElective, progElectCount, progENCount, saveprogram, restoreprogram, newStudent, enrolProg, enrolCourse, disenrolCourse, completeCourse0, failCourse0, completeProg0, moreStudents, validStudent, studentEnrolled, inProgram, courseEnrolled, courseCompleted, prereqsCompleted, coreCompleted, electiveCompleted, genedCompleted, coursesEnrolled, programEnrolled, anyStudent, savestudent, restorestudent

PROPERTIES
ASSESSOR = ASSESSORDB_ABSOBJ ∧
COURSENAME = STRING ∧
PROGNAME = STRING ∧
ASSESSORNAME = STRING

INVARIANT
ASSESSORDB = assessors ∧
AssessorName = assessorname ∧
∀ as . ( as ∈ ASSESSORDB ⇒
  Assesses ( as ) = assesses [ { as } ] )

OPERATIONS
newAssessor ( asname ) ≡
  VAR bb, db IN
  bb, db ← make_ASSESSORDB ;
  bb ← mod_AssessorName ( db, asname )
END ;
addAssessor ( assessor, course ) ≡
  VAR bb IN
  bb ← add_Assesses ( assessor, course )
END ;
delAssessor ( assessor, course ) ≡
BEGIN
del_Assesses ( assessor, course )
END ;
completeCourse ( assessor, student, course ) ≡
BEGIN
  completeCourse0 ( student, course )
END ;
failcourse ( assessor, student, course ) ≡
BEGIN
  failCourse0 ( student, course )
END ;
ok $\leftarrow$ MoreAssessors $\triangleq$

$$\text{VAR } \text{tot IN}$$

$$\text{tot } \leftarrow \text{nbr\_ASSESSORDB} ;$$

$$\text{IF } \text{tot } \neq \text{maxAssessor } \text{THEN}$$

$$\text{ok } := \text{TRUE}$$

$$\text{ELSE } \text{ok } := \text{FALSE}$$

$$\text{END}$$

$$\text{END} ;$$

ok $\leftarrow$ newAssessorName( name ) $\triangleq$

$$\text{VAR } \text{bb , db IN}$$

$$\text{bb , db } \leftarrow \text{key\_search\_AssessorName ( name )} ;$$

$$\text{ok } \leftarrow \text{NEG\_BOOL ( bb )}$$

$$\text{END} ;$$

ok , assessor $\leftarrow$ validAssessor( asname ) $\triangleq$

$$\text{BEGIN}$$

$$\text{ok , assessor } \leftarrow \text{key\_search\_AssessorName ( asname )}$$

$$\text{END} ;$$

ok $\leftarrow$ AssessorAccredited( assessor , course ) $\triangleq$

$$\text{BEGIN}$$

$$\text{ok } \leftarrow \text{member\_Assesses ( assessor , course )}$$

$$\text{END} ;$$

ok $\leftarrow$ saveassessor $\triangleq$

$$\text{VAR } \text{bb , cc IN}$$

$$\text{bb } \leftarrow \text{save\_genered} ;$$

$$\text{cc } \leftarrow \text{save\_AssessorDB} ;$$

$$\text{ok } \leftarrow \text{CNJ\_BOOL ( bb , cc )}$$

$$\text{END} ;$$

ok $\leftarrow$ restoreassessor $\triangleq$

$$\text{VAR } \text{bb , cc IN}$$

$$\text{bb } \leftarrow \text{restore\_student} ;$$

$$\text{cc } \leftarrow \text{restore\_AssessorDB} ;$$

$$\text{ok } \leftarrow \text{CNJ\_BOOL ( bb , cc )}$$

$$\text{END}$$

$$\text{END}$$

### B.6 Implementation of ProgramManager

**IMPLEMENTATION** ProgramManagerI  
**REFINES** ProgramManager  
**SEES**  
AssessorName , Assessor\_CTX , 
Student\_CTX ,  
ProgName , Program\_CTX ,  
CourseName , Course\_CTX ,  
Bool\_TYPE , Bool\_TYPE\_Ops , String\_TYPE

**IMPORTS**  
Assessor ( maxAssessor , maxStudent , 
maxProg , maxCourse , maxPrereqs )
PROMOTES
newCourse , addPrereq , delPrereq , firstPreReq ,
nextPreReq , MoreCourses , anyCourse , newCourseName ,
CourseExists , courseName , InPrereqs , ACyclicPrereq ,
HasPrereqs , MorePrereqs , newgeCourse , delgecourse ,
isGenEd , savegened , restoregened , newProgram , addCore ,
delCore , addElective , delElective , setElectCount ,
setGECount , progName , MorePrograms , newProgName ,
ProgramExists , isCore , isElective , firstCore ,
nextCore , firstElective , nextElective , progElectCount ,
progGECount , saveprogram , restoreprogram , newStudent ,
enrolProg , enrolCourse , disenrolCourse ,
completeCourse0 , failCourse0 , completeProg0 ,
moreStudents , validStudent , studentEnrolled ,
inProgram , courseEnrolled , courseCompleted ,
prereqsCompleted , coreCompleted , electiveCompleted ,
genedCompleted , coursesEnrolled , programEnrolled ,
anyStudent , savestudent , restorestudent , newAssessor ,
addAssessor , delAssessor , completeCourse , failcourse ,
MoreAssessors , newAssessorName , validAssessor ,
AssessorAccredited , saveassessor , restoreassessor

PROPERTIES
COURSENAME = STRING ∧
PROGNAME = STRING ∧
ASSESSORNAME = STRING

OPERATIONS

completeProg( student ) ≡

VAR program IN

program ← programEnrolled ( student ) ;
completeProg0 ( student , program )

END ;

ok ← ProgramCompleted ( student ) ≡

VAR bb , cc , ec , gc IN

cc ← coreCompleted ( student ) ;
ec ← electiveCompleted ( student ) ;
gc ← genedCompleted ( student ) ;
bb ← CNJ_BOOL ( cc , ec ) ;
ok ← CNJ_BOOL ( bb , gc )

END

END

B.7 Implementing SEEN machines

All machines must be implemented, including SEEN machines. These are given below.

IMPLEMENTATION Course_CTXI
REFINES Course_CTX
END

IMPLEMENTATION CourseName1
REFINES CourseName
SEES String_TYPE, String_TYPE_Ops
PROPERTIES COURSENAME = STRING

OPERATIONS
  csname ← anyCOURSENAME ≡
    csname ← CPY_STR ("NONAME")
END

IMPLEMENTATION Program_CTXI
REFINES Program_CTX
END

IMPLEMENTATION ProgNameI
REFINES ProgName
SEES String_TYPE, String_TYPE_Ops
PROPERTIES PROGNAME = STRING

OPERATIONS
  prname ← anyPROGNAME ≡
    prname ← CPY_STR ("NONAME")
END

IMPLEMENTATION Assessor_CTXI
REFINES Assessor_CTX
END

IMPLEMENTATION AssessorNameI
REFINES AssessorName
SEES String_TYPE, String_TYPE_Ops
PROPERTIES ASSESSORNAME = STRING

OPERATIONS
  prname ← anyASSESSORNAME ≡
    prname ← CPY_STR ("NONAME")
END